12-2016

Securing cloud-based data analytics: A practical approach

Julian James Stephen
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Julian James Stephen

Entitled

Securing Cloud-Based Data Analytics: A Practical Approach

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Patrick Eugster / Sandra Freeman
Co-chair

Aniket Kate
Co-chair

Dongyan Xu

Mathias Payer

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Aniket Kate

Approved by: Sunil Prabhakar / William J. Gorman 12/5/2016

Head of the Departmental Graduate Program Date

SECURING CLOUD-BASED DATA ANALYTICS: A PRACTICAL APPROACH

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Julian James Stephen

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

To my family.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Patrick Eugster for the continuous support of my Ph.D study and related research, for his patience, motivation, and knowledge.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Aniket Kate and Prof. Dongyan Xu for their insightful comments and encouragement.

I thank my collaborators for the wonderful discussions, sleepless nights before deadlines, and for all help and feedback during the last many years.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

James Stephen, Julian PhD, Purdue University, December 2016. Securing Cloud-Based Data Analytics: A Practical Approach. Major Professor: Patrick Eugster.

The ubiquitous nature of computers is driving a massive increase in the amount of data generated by humans and machines. The shift to cloud technologies is a paradigm change that offers considerable financial and administrative gains in the effort to analyze these data. However, governmental and business institutions wanting to tap into these gains are concerned with security issues. The cloud presents new vulnerabilities and is dominated by new kinds of applications, which calls for new security solutions. In the direction of analyzing massive amounts of data, tools like MapReduce, Apache Storm, Dryad and higher-level scripting languages like Pig Latin and DryadLINQ have significantly improved corresponding tasks for software developers. The equally important aspect of securing computations performed by these tools and ensuring confidentiality of data has seen very little support emerge for programmers.

In this dissertation, we present solutions to a. secure computations being run in the cloud by leveraging BFT replication coupled with fault isolation and b. secure data from being leaked by computing directly on encrypted data. For securing computations (a.), we leverage a combination of variable-degree clustering, approximated and offline output comparison, smart deployment, and separation of duty to achieve a parameterized tradeoff between fault tolerance and overhead in practice. We demonstrate the low overhead achieved with our solution when securing data-flow computations expressed in Apache Pig, and Hadoop. Our solution allows assured computation with less than 10 percent latency overhead as shown by our evaluation. For securing data (b.), we present novel data flow analyses and program

transformations for Pig Latin and Apache Storm, that automatically enable the execution of corresponding scripts on encrypted data. We avoid fully homomorphic encryption because of its prohibitively high cost; instead, in some cases, we rely on a minimal set of operations performed by the client. We present the algorithms used for this translation, and empirically demonstrate the practical performance of our approach as well as improvements for programmers in terms of the effort required to preserve data confidentiality.

1. INTRODUCTION

The cloud as a computing platform is getting more popular and mature every day. Computational needs of industry, academia, and government are being increasingly met by processing data in the cloud. Recently announced government policies [1] clearly show an urgent economic requirement for processing data in the cloud. Yet, a major roadblock to adopting cloud technologies is the lack of trust on the various facets of cloud computing. The fact that a potentially malicious entity can legally access computing resources in the same data center or even on the same machines as well-intended users increases the risk associated with moving computations into the cloud. Malicious programs, faulty hardware, and software bugs can cause services to fail and leak confidential data. In this context, it is important to have solutions that allow us to leverage the immense potential of the cloud without sacrificing security of computation and data.

1.1 Security in the Cloud

The term "security" is very broad and defines many criteria and corresponding goals. We limit our discussion to the security goals of confidentiality, integrity and availability. Most approaches to cloud security geared towards these goals focus by and large on either (a) *communication*, (b) *data*, or (c) *computation*. Communication-centric approaches (a) to security in public or in-house clouds focus on setting up thick firewalls, which monitor in - and outbound traffic. Typically, ports that accept incoming data and specific protocols and services are allowed or disallowed based on the configuration of the firewall. Though required, such perimeter security is not sufficient because, zero-day attacks or insider attacks may compromise one or many of the internal nodes. Once an internal node is compromised, it can alter computation

output, delay computations and read confidential data. This can break the integrity, availability and confidentiality of the system. Data-centric approaches (b) protect data from computation failures and confidentiality leaks. In all functional systems, data is under constant churn. Computation adds, deletes and morphs data into new forms. From a security perspective, this means that efforts to secure data should not make data non-pliable. Computation-centric approaches (c) to securing computation focus on fine-grained information-flow [2]. As with data-centric approaches, information-flow approaches aim at protecting (sensitive) data from leaking. However, they do not ensure that the computation is behaving according to specification, i.e., ensuring the computation is doing what it was intended to. In typical cloud data-flow processing applications, where new data-sets are generated as outcome of analysis and correlation of existing data-sets and stored for subsequent use, these outcomes must be trustworthy. In fact, since in the larger picture data-sets are derived from earlier data-sets, any false results computed violate integrity of the semantic information in the original data-sets.

1.2 State of the Art

For ensuring the security goals of integrity and availability in the presence of arbitrary faults, *Byzantine fault tolerant* (BFT) replication [3] can be utilized. BFT techniques rely on multiple replicas of a computation comparing their output (assuming a correct "majority") to weed out erratic behaviour. While several fundamental assumptions of BFT replication — e.g., determinism in replicas for comparisons, possibility of exploiting redundant hardware — are largely met by typical cloud-based data-flow applications, *existing* BFT *systems* are inapplicable to such applications: these focus on securing single monolithic servers, and only little work exists on applying BFT replication beyond such stand-alone servers. Cloud-based data-flow processing systems, inversely, leverage cheap hardware by breaking down applications into small components which are amenable to parallel execution. When applying BFT

replication to any one of these components by running multiple replicas of each and comparing their respective outcomes overheads sum up very quickly.

For ensuring confidentiality, data can be encrypted with cryptographic algorithms. Further, several existing cryptographic systems (cryptosystems) allow meaningful operations to be performed directly on encrypted data. Advances occurring regularly in fully homomorphic encryption (FHE) schemes further increase the scope and performance of computations on encrypted data. Unfortunately, such advancements in cryptography have not translated into programmer-friendly frameworks for big data analytics with acceptable performance. Currently, for big data programmers to efficiently put existing homomorphic schemes to work they will have to explicitly make use of corresponding cryptographic primitives. To make this task even more difficult for the programmer, existing big data analytic jobs are typically expressed in domain specific security-agnostic high-level data flow languages. This makes the explicit use of cryptographic primitives even harder.

1.3 Contributions

In this dissertation we investigate approaches for securing analytic computations on big data that run in cloud-like data centers. We start by investigating mechanisms for ensuring integrity and availability of a sequence of tasks (e.g., MapReduce jobs) in the presence of arbitrary (Byzantine) faults. We explore replication based techniques and develop a system that allow configurable trade-offs between performance and correctness. We leverage the information we obtain from the execution of replicas to attribute fault and further isolate malicious nodes.

We then explore mechanisms that ensure confidentiality of data used for analytics and prevent data from being *leaked*. To do this we develop systems for batch and stream based computations. For both systems, we leverage properties of partial homomorphic cryptographic systems that allow a minimal set of computations to be executed on data that is already encrypted. We use program analysis to transform

programs written for non encrypted data sets or data streams and convert them to programs that work on encrypted data.

1.4 Problem Statement

There are currently no practical, programmer friendly solutions for securing cloud based data analytics against arbitrary faults or data leaks. Current solutions place an impractical burden on the programmer, are costly and does not offer solutions that fit the cloud computing model.

1.5 Thesis Statement

This thesis will introduce systems for securing distributed, cloud based data analytics. These solutions will use replication to ensure integrity and availability, and computations over encrypted data to ensure confidentiality.

1.6 Organization

This dissertation contains six chapters. Chapter 2 describes our system for securing integrity and availability of computations using Byzantine fault tolerant replication. Chapter 3 describes our system for ensuring confidentiality of data analytic jobs. Chapter 4 also focuses on ensuring confidentiality, but looks specifically at the challenges posed by stream data analytics. Chapter 5 discusses related work and Chapter 6 concludes the dissertation.

2. INTEGRITY AND AVAILABILITY IN THE CLOUD

This chapter presents our solution for performing byzantine fault tolerant data analytics (published in [4]).

2.1 Overview

Over the past few years, compute clouds have emerged as cost-efficient big data analysis platforms for corporations and governments. Yet, many organizations still decline to widely adopt cloud services because threats from zero day attacks or account hijacking can result in compute nodes being compromised and integrity of computations being violated. Intuitively, *Byzantine fault tolerant* (BFT) replication [3] is a powerful means of securing computation and thus achieving integrity and availability in cloud-based computing. BFT suggests the use of multiple replicas of a sensitive component, and hinges on the comparison of outcomes produced by these replicas to determine components with erratic behavior (assuming a correct "majority"). While several fundamental assumptions of BFT replication — e.g., determinism in replicas for comparisons, possibility of exploiting redundant hardware — are largely met by typical cloud-based data-flow applications, *existing* BFT *systems* are inapplicable to such applications: these focus on securing single monolithic servers, and only little work exists on applying BFT replication beyond such stand-alone servers. Cloud-based data-flow processing systems, inversely, leverage cheap hardware by breaking down applications into small components which are amenable to parallel execution. When applying BFT replication to any one of these components by running multiple replicas of each and comparing their respective outcomes overheads sum up very quickly.

2.1.1 Model

In many scenarios, institutions can trust the cloud providers themselves, but not the users of the system. If we take the example of the US intelligence community, different agencies have their own *inhouse* clouds. They want to improve sharing of information with each other without exposing their own systems to potential weaknesses or infections in their peer systems [1]. Such a partial trust model also applies to many large corporations which include subdivisions hosting their own datacenters. Within this scenario, the present chapter is concerned with ensuring (a) *integrity* and (b) *availability*, i.e., that computations indeed perform what they were supposed to (e.g., to avoid obfuscating terrorist activities), and that these computations can be performed in a timely manner (e.g., to be able to react to real threats on time).

2.1.2 ClusterBFT

This chapter presents ClusterBFT for cloud-based assured data processing and analysis. ClusterBFT utilizes BFT replication techniques which impose less overhead than existing cryptographic primitives, but breaks away from the mold of individually replicating every client request. More precisely, ClusterBFT creates sub-graphs from acyclic data-flow graphs that are then replicated. This means, rather than enduring the overhead of BFT consensus at each component involved in the data-flow processing, we have a system with much less overhead that can dynamically adapt to changes in required responsiveness and perceived threat level as well as to dynamic deployment (elasticity). We use a combination of *variable-grain clustering* with *approximated* and *offline comparison*, *separation of duty*, and *smart deployment* to keep overheads of BFT replication low while providing good fault isolation properties. In summary, the main contributions of the chapter are (1) identification of challenges and solutions for achieving availability and integrity of cloud-based data-flow computations with BFT replication, (2) the architecture and implementation of a BFT solution for such computations, and (3) the evaluation of this solution. Our evalua-

tion shows less than 10 percent latency overhead in most cases for even complex data analysis jobs.

2.1.3 Roadmap

The remainder of this chapter is structured as follows. Section 2.2 provides background information. Section 2.3 lists design principles and challenges. Section 2.4 describes the ClusterBFT architecture in detail. Section 2.5 describes its implementation. Section 2.6 presents evaluation results. Section 5.1 presents related work.

2.2 Background and Preliminaries

This section presents information pertinent to the remainder of the chapter.

2.2.1 BFT

Byzantine failures [3] model arbitrary faults that may occur in a process during execution, including malign and benign faults. In order to explain our system better, we further distinguish Byzantine failures by classifying them based on how they allow deviation from correct execution. We use the categorization of Kihlstrom et al. [5] which classifies Byzantine failures as follows:

- Omission (detectable): An omission failure occurs when a process does not send a message that it is expected to send. These can be detected by setting timeouts for messages. It is important to note here that in an asynchronous system, a timeout does not necessarily imply a faulty component.

- Commission (detectable): A commission failure occurs when a process sends a message it is not supposed to send. Such failures can be detected by checking if the message is in agreement with at least *f+1* other replicas.

- Unobservable (non-detectable): Unobservable failures are those which other processes cannot detect based on the messages they receive.
- Undiagnosable (non-detectable): Undiagnosable failures are those that cannot be attributed to a specific process.

2.2.2 MapReduce and Pig

Big data analysis is one of the major use cases for moving towards cloud computing and most cloud-specific programming models reflect this. Corresponding runtime systems try to make use of large numbers of nodes available for data analysis to decrease latency. The popular MapReduce [6] framework partitions input data and assigns a mapper process to each input partition. These mapper processes produce "intermediate" key-value pairs as output which are grouped by key and fed by key to reducer processes which use these to generate final output. Hadoop [7] is a popular open source implementation of MapReduce.

Apache Pig [8] is a platform for data analysis that consists of the PigLatin [9] high-level language for expressing data analysis programs, and a runtime system. Pig Latin scripts are typically compiled to MapReduce jobs that are executed using a MapReduce engine such as Hadoop for Pig. To illustrate the benefits of our concepts, we focus in this chapter on Pig data analysis jobs.

Throughout the chapter, unless otherwise specified, we use the term *script* to refer to a Pig (Latin) script. We use the term *job* to refer to a MapReduce job and *task* to refer to map or reduce tasks within a MapReduce job. We use the term *job cluster* to refer to the group of nodes involved in executing a specific job.

2.2.3 System Model

We assume that the system is deployed on a cloud service that leases out virtual machines to users. We refer to one such virtual computation unit as a *node*. This

means that there could be multiple nodes on the same physical machine. We assume that the number of nodes that are faulty at a given time is bounded. For the purpose of this chapter we focus on computation and assume a trusted storage layer. We are aware that assuming correctness of a storage system prone to Byzantine faults is ambitious, but it is not unrealistic either. Systems like DepSky [10] show its feasibility. Further, the challenges that need to be solved even with the presence of a trusted storage are tough and warrant investigation. We present a system for two *adversary models*. For both models, we assume that the adversary cannot manipulate the cloud service provider or violate its specifications. This includes preventing communication between any two nodes, spawning new Byzantine nodes, and breaking computationally hard cryptographic primitives. A *strong adversary* can manipulate all internal aspects of a node and collude with other adversaries. This includes full control over the executing processes, physical memory and messages being sent out of the node. A *weak adversary* shares the same properties of a strong adversary, but may only cause omission or commission faults.

2.3 ClusterBFT Design

This section presents first our motivation for using BFT techniques in the cloud, before outlining challenges in such adoption and finally our solutions for overcoming these.

2.3.1 BFT and the Cloud

We decided to adopt BFT replication due to several intuitive benefits:

Attribution: Along with tolerating benign or malign failures, BFT techniques can also point to potentially faulty components which helps for attribution as well as auditing. Indeed, being able to shield computation from malicious entities is one thing, but in a sea of nodes such as a cloud datacenter it is also necessary to

keep track of where such accesses were attempted, as these may hint to exploited leaks and intruders.

Portability and interoperability: BFT techniques can be applied at a higher level in the protocol stack — here typically at the level of data-flow program execution — which allows them to be deployed easily across different cloud platforms and infrastructures, thus supporting portability, cloud interoperability, and the cloud-of-clouds paradigm [11].

Determinism: Popularity of data-flow languages like PigLatin or DryadLINQ [12] shows the relevance of data analysis jobs that can be modeled as direct acyclic graphs (DAGs). These computations and their constituents are by-and-large deterministic, which simplifies the comparison of redundant results. Inversely, concurrent client accesses pose challenges when replicating large monolithic servers. Recent trends in cloud-based data processing include support for iterative and incremental jobs which contradict the straightforward DAG model [13] but do not hamper determinism.

Heterogeneity: BFT replication relies on heterogeneity of replicas to ensure that a majority of replicas are not compromised simultaneously by means of the same vulnerability. Cloud platforms do expose a uniform hypervisor layer on which operating systems are deployed, but cloud providers offer a variety of operating system images that can be deployed on these nodes. Within an operating system itself, adoption of address space layout randomization (ADSLR) introduces further heterogeneity. DARPA's Mission-Resilient Cloud program [14] funds several projects aiming at creating moving targets specifically to further narrow this gap [15].

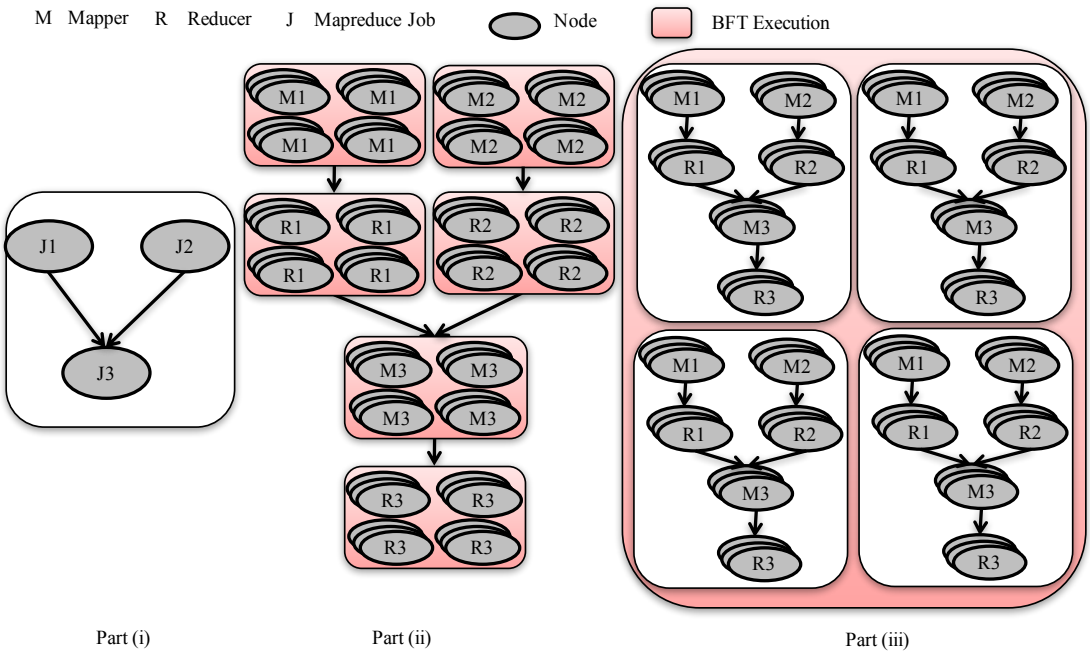


Fig. 2.1.: Part (i) shows a data-flow graph with 7 phases. (ii) focuses on $n \times m$ replication of jobs J1, J2 and J3. (iii) shows clustered replication of J1, J2, J3 requiring only one round of BFT consensus. Tasks within a MapReduce job are shown one behind the other.

2.3.2 Challenges in Adopting BFT in the Cloud

Though intuitively BFT replication seems like a good match in many ways for ensuring computation in the cloud, it has thus far not been adopted widely in such a context due to a variety of open challenges:

C1. *Scalability:* Datasets are typically many magnitudes higher in cloud-based programming than in previous scenarios. As BFT replication protocols hinge on comparison of redundant outcomes, this translates to large overheads.

C2. *Granularity:* Data analysis scripts also tend to have multiple *jobs* where output of one is fed to the second. This creates a *job-chain* in which a process that was a server for one job acts as a client for the second job. Ideally, every process is fine-grained and can be deployed dynamically. This means, naïve

BFT replication of each job will result in $R = 3f + 1$ replicas for each task, with $n \times m$ communication [16] and synchronization after every stage. This is illustrated by Figure 2.1. The left part (i) shows a Pig-style data-flow graph, while the middle part (ii) illustrates the $n \times m$ interaction [16] occurring as a result of replicating every node in the (sub)graph obtained after compilation to MapReduce jobs: every edge corresponds here in fact corresponds to 4×4 interactions. This causes very high resource usage, limiting availability and increasing cost for huge data-flows.

C3. *Rigidity:* Clouds represent very dynamic environments, being marketed to meet instantaneous demands rather than having to over-provision constantly to meet occasional spikes. This calls for solutions that are flexible and can be adapted to some degree. The main knob to turn in BFT replication is the replication degree, which however represents a coarse granularity: typically a replication degree of $3f+1 = 4$ with $f = 1$ already leads to substantial overhead. The next larger step, $3f+1 = 7$, to tolerate up to 2 failures already leads to prohibitively larger overhead.

There are are also non-technical factors deterring adoption of BFT replication in the cloud. As explained by Birman et al. [17], many cloud middleware service providers have an inherent "fear of synchronization" irrespective of the existence of fast consensus protocols and success stories like Chubby [18].

2.3.3 ClusterBFT Principles and Architecture Overview

ClusterBFT addresses the challenges C1-C3 above as follows:

Variable granularity: Observe that nothing forces us to replicate every individual node in the data-flow graph. We could in fact replicate the execution of an entire data-flow graph $3f + 1$-fold, and compare the outcomes at the very end. More generally speaking, we can choose any intermediate level for clustering

nodes in the graph and replicate these subgraphs (addressing C2 above). This is illustrated by the right part (iii) of Figure 2.1, where the sub-graph of (ii) is replicated as a whole and comparison only occurs at the end of this sub-graph. The potential downside of such regrouping is that it may diminish the degree of fault tolerance and precision of fault attribution: a single deviant node in a group hampers the outcome of that replica, and identifying *which* node(s) in the group exhibit Byzantine behavior becomes harder. In addition, if we do not end up having sufficiently many identical replica responses, it takes longer to run additional replicas thus increasing job latency. This tradeoff leads to an additional knob for users to tweak (C3).

Variable replication: The BFT replication model allows control over how resources are utilized. Based on the user's confidence in the cluster, different degrees of replication can be adopted with different guarantees. A user can specify an optimistic, $f + 1$ replicas. In this case, the execution ensures safety, but may require repeated runs to get correct output. If the user specifies $2f + 1$ replicas, a correct result can be guaranteed if all replicas always reply (no omission failures). If $3f + 1$ replicas are specified, a correct result can be guaranteed under combination of any kind of Byzantine failure.

Approximate, offline redundancy: Instead of comparing the entire outputs of a replica set in one go upon sub-job completion, we can choose to (1) only compare *digests*, (2) start doing so *before sub-job completion*, and (3) allow the follow-up sub-job to proceed based on the complete output *before comparison completes*. This reduces the overhead of putting redundancy to work (addressing C1) in a way allowing further fine-tuning of tradeoffs between performance and security by control of the resilience of the digests (C3).

Separation of duty: Rather than baking the entire data-flow handling logic into every node, we can separate architecturally the "front-end" of a data-flow processing system which accepts jobs from the actual cluster of worker nodes such as

MapReduce nodes executing the jobs. This architectural division is illustrated in Figure 2.2 which outlines the architecture of our solution ClusterBFT detailed in the next section. Components in the *control* tier are command and control processes that provide direction and coordinate computations in the *computation* tier. The former tier is trusted, which is achievable by BFT replication or by implicitly trusting the nodes, i.e., by closely (even manually) monitoring nodes, or using nodes in the client network or private cloud. The benefit of this separation is that it limits certain strong assumptions and expensive mechanisms to the front-end, allowing the cluster to focus on work (cf. [19]) and to be handled more dynamically (C3). This in turns allows the worker node cluster to be adapted dynamically, by adding and removing nodes based on resource requirements, measured performance, and of course suspicions.

Fault isolation: Another net advantage of the separation of duty is that the front-end can keep track of suspicions observed, and can use specific deployment policies to, for instance, narrow down the (set of) faulty node(s) in a replication group delivering a faulty response by intentionally partly overlaying the replication group of a different job on the same nodes. Similarly, dummy jobs can be used to further probe nodes in such a suspicious replication group. Thus the tradeoff with attribution precision introduced by variable granularity does not become a one-way path but becomes a tradeoff with the *time* it takes to recover precision (C3).

In the following section we describe the architecture of ClusterBFT and how we put these principles to work.

2.4 ClusterBFT Architecture and Components

In this section we look at the different components that make up ClusterBFT (see Figure 2.2). Table 2.1 presents a summary of the symbols used in the following.

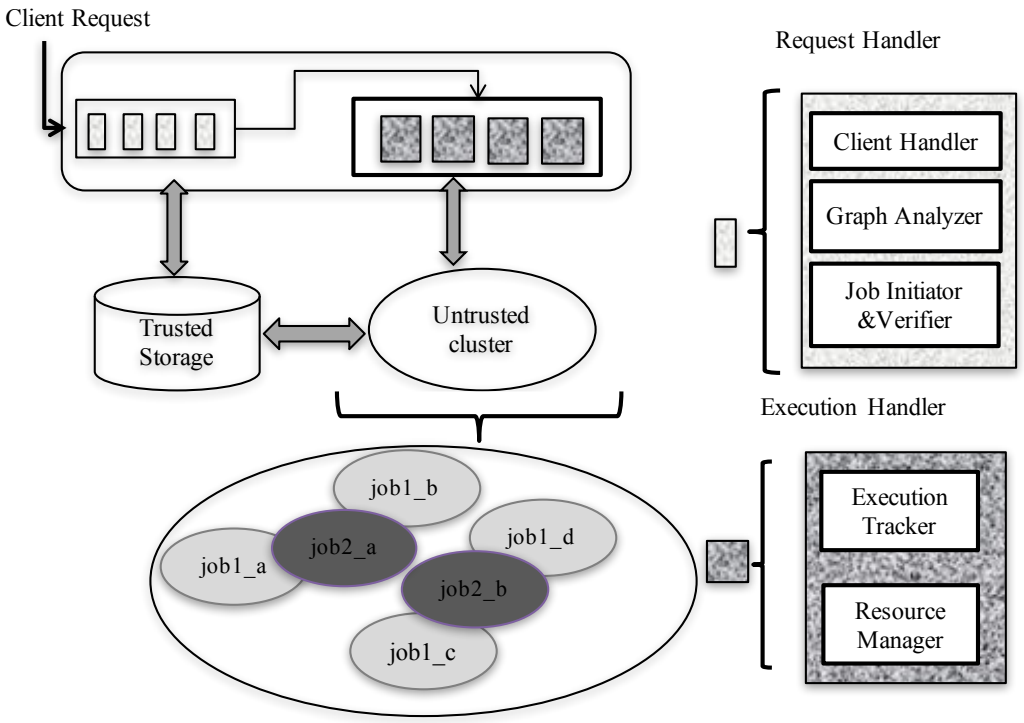


Fig. 2.2.: Architecture

2.4.1 Request Handler

The *request handler* component is in the control tier. It accepts scripts submitted by the client and submits the script for execution. It consists of three logical subcomponents outlined below.

Client handler. The client submits the script to the *client handler*. The client also specifies the number of expected failures f, a replication factor r and the total number of verification points n based on the perceived threat level. Verification points are vertices in the data-flow graph after which output from different replicas are matched. The client handler generates a logical plan from the script. This is given as input to the graph analyzer described below.

Graph analyzer. In order to reduce overhead and improve utilization, we need to identify verification points in the data-flow graph that are most effective. Running verification after every operation will cause very high overhead and running veri-

Table 2.1.: Symbols

Symbol	Meaning
r	Replication factor
n	Number of verification points
f	Number of expected failures
s	Suspicion level

Table 2.2.: Notation

Notation	Meaning
$ir[v]$	Input ratio of v
$parents(v)$	All parents of vertex v
$level(v)$	$\begin{cases} 1 & \text{if } v = Load \\ \max\limits_{p \in parents(v)} 1 + level(p) & \text{else} \end{cases}$
$min(v, M)$	Number of edges between v and the vertex closest to v in M

fication scarcely will result in more re-computations (hence higher resource usage) when failures occur. The *graph analyzer* component, based on the adversary model, identifies points in the data-flow graph for performing verification. Under the strong adversary model, only points that correspond to data-flow between jobs are considered for verification. Under a weak adversary model, any point in the data flow graph can be considered for verification.

With n verification points requested by the user, we use the marker function defined in Algorithm 1 to identify the actual points. The symbols and notations used in Algorithm 1 are defined in Tables 2.1 and 2.2. We explain the intuition behind the marker function using an example. Consider the data-flow graph in Figure 2.3 and assume the user specified one verification point. If we decide to perform verification right after the vertex $Load1$, then the probability of identifying a fault is very low. There is a much higher probability that at least one of the nodes that execute the vertices below $Load1$ is faulty simply because there are more of them. On the other end, if we run verification after $Join2$, then we most probably will know if result

Algorithm 1 Marker function

```
1: V                                                          ▷ Vertex set
2: n                                          ▷ Number of verification points
3: function MARK(V, n )
4:     M ← ∅                                          ▷ Set of marked vertices
5:     for 1.. n do
6:         max ← 0
7:         for all v ∈ V do
8:             score_v ← ir[v] + min(v, M)
9:             if score_v > max then
10:                m ← v
11:                max ← score_v
12:            end if
13:        end for
14:        M ← M ∪ {m}
15:    end for
16: end function
17: function INPUT_RATIO(v)
18:    if v is Load then
19:        ir[v] ← input_size(v) / total_input_size
20:    else
21:        ir[v] ← ( Σ_{p∈parents(v)} ipr[p] ) / ( Σ_{level(n)=level(v)−1} ipr[n] )
22:    end if
23: end function
```

is going to be faulty, but the cost of re-computation, in case $f + 1$ replicas do not agree becomes high; the entire sequence of operation needs to be recomputed. The marker function considers two main parameters to arrive at a verification point that is a good tradeoff between these two extremes. The ratio of input data that flows through a vertex and distance of a vertex from another verification point. Using these two values, the marker function arrives at a mid point suitable for verification. Once the verification point is identified, the logical plan is instrumented with a verification function and given to the *job initiator*. Details of what a verification function are described next.

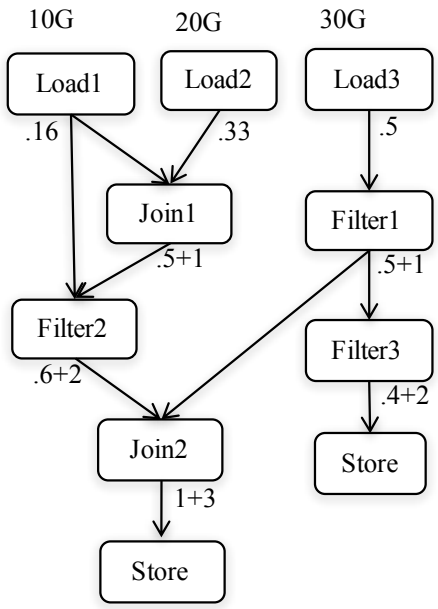


Fig. 2.3.: Annotated data-flow graph

Job initiator and verifier. The instrumented script gets compiled into one or more MapReduce jobs and the job initiator associates a sub graph identifier *sid* with each such job. The job initiator submits a total of r replicas of the job for execution to the execution handler. All replicas are configured to have the same number of reduce tasks. The verification function instrumented into the MapReduce job uses a cryptographic hash function (SHA-256 in our prototype) to compute a digest of the data streaming through the verification point and sends this digest to the verifier. The verifier compares corresponding digests from different replicas and asserts that at least $f + 1$ are same. The verifier is also responsible for isolating failures and updating the suspicion level s for each node.The suspicion level of a node is defined as total number of faults associated with the node divided by the total number of jobs executed on the node. For clarity, details of fault isolation is specified as a separate section (2.4.3), after we introduce the remaining components in our architecture.

2.4.2 Execution Handler

Figure 2.4 shows the internals of the *execution handler* and how it interacts with the request handler.

Execution tracker. The job submitted by the request handler is executed by the execution tracker. Resources available in nodes are partitioned into uniform resource units ru. A list of all resources is initially loaded from an administrator-provided inclusion list into the *resource table* as a tuple $\langle nid, \#ru, \langle sid... \rangle, s \rangle$. One tuple represents a node id nid, the number of resource units ru in that node, the current allocation of sids and suspicion level s of a node. When the job initiator submits a job, the job is first added to the job queue. The main sequence of operations that take place after this is shown in Figure 2.4 (others are omitted for simplicity), and detailed below:

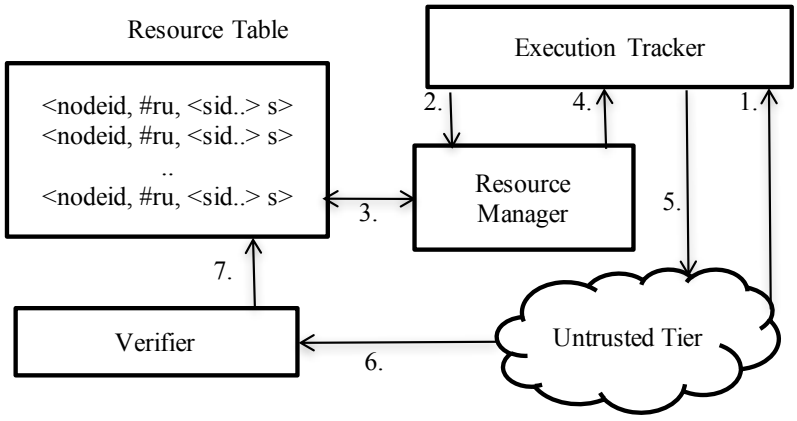


Fig. 2.4.: Execution tracker & resource manager

1. A node in the untrusted domain with id nid sends a heartbeat message to the execution tracker.

2. The execution tracker checks with the *resource manager* to see if there is a task that can be scheduled on node nid.

3. The resource manager queries the resource allocation table to retrieve the *sid*s of tasks currently running on node *nid*. Using this, the resource manager looks at the list of running or submitted jobs to check if a there is a task from a job that does not already have a task running on node *nid*.

4. The resource manager provides a list of ready tasks corresponding to the number of free *ru*s in node *nid*.

5. The execution tracker replies to the heartbeat message with the task that needs to be executed.

6. During task execution, the verification function creates a message digest of data streaming through the verification point and sends the digest to the verifier. The verifier checks for $f + 1$ matching message digests from different replicas. If the verifier times out without obtaining $f + 1$ matching message digests, the job is initiated again with a higher value for r.

7. Based on the number of non-matching digests, the verifier updates the suspicion levels of node *nid*. in the resource table.

Resource manager. We already outlined how the resource manager functions as part of the working of the execution tracker. There are two goals that we try to achieve by proper task selection: efficient execution and fast fault identification. Data local tasks enable faster execution. For fast fault identification job clusters can be overlapped in specific patterns. The scheduling strategy we use is to cause as many intersections as there are resource units in a node. That means if one node has three resource units, we try to pick tasks from three different jobs to execute. Other strategies can also be used to overlap clusters which we intend to explore in future work. The administrator can also configure a suspicion threshold such that if $s >$ threshold, then the resource manager will remove that node from its inclusion list and ignore further requests from that node. At this point administrators can intervene to

re-initialize the node by taking the node off the grid, applying securing patches and reinserting the node.

Algorithm 2 Fault analyzer function

```
1: D ← A set of disjoint sets.
2: O ← A set of overlapping sets.
3: function FAULT ANALYZER(S)                ▷ S, set of nodes with commission fault
4:     if ∀X | X ∈ D, S ∩ X = ∅ then
5:         D ← {S} ∪ D
6:     else if ∃Y | Y ∈ D and S ⊂ Y then
7:         D ← D \ {Y}
8:         O ← O ∪ {Y}
9:         D ← D ∪ {S}
10:    else
11:        O ← O ∪ {S}
12:    if |D| = f then
13:        for each X ∈ D do
14:            A ← A ∪ X
15:        for each X ∈ O do
16:            X ← X ∩ A
17:        for each X ∈ D do
18:            for each Y ∈ O do
19:                if X ∩ Y ≠ ∅ then
20:                    I ← I ∪ (X ∩ Y)
21:            if | I |= 1 then
22:                D ← D \ {X}
23:                D ← D ∪ {I}
24: end function
```

2.4.3 Fault Identification and Isolation

As described in Section 2.4.2, the output verifier collects output digests and asserts that at least $f + 1$ digests are the same. If the verifier receives an incorrect digest or does not receive a digest from nodes executing the data-flow, the suspicion level of all involved nodes is updated. This means if there is a faulty node that is part of multiple job clusters, that faulty node is likely to have a higher suspicion level. Once the verifier identifies a job cluster as returning incorrect result, the *fault analyzer*

function in Algorithm 2 is used to further narrow down the list of suspicious nodes. The fault analyzer works in two stages. In the first stage disjoint subsets of suspicious nodes are isolated. This set of subsets is denoted by D (line 1). This is done until the number of such subsets becomes equal to the highest value of f the system has seen so far (line 12). This allows us to identify subsets of nodes such that there is exactly one fault per subset. The second stage (lines 13-23) reduces the number of nodes in these subsets by creating the intersection of a subset with other sets of faulty nodes. The intuition for the second stage is that if there are f subsets in D and a new set of faulty nodes intersects with only one of those f subsets, then the nodes in the intersection must be faulty.

Byzantine behavior also means an infected node may be mostly producing correct output, and produce incorrect results occasionally. This means if nodes show malicious intent/fail frequently, fault isolation becomes faster.

2.5 Implementation

This section presents our prototype implementation of ClusterBFT. ClusterBFT is implemented in Java by modifying Hadoop 1.0.4 [20] and Pig 0.9.2. For instrumenting the Pig logical plan we modified the Penny [21] monitoring tool, distributed as part of Pig 0.9.2 source.

2.5.1 Hadoop

Hadoop uses a centralized *job tracker* and *task tracker*s on each computation node. The job tracker initiates a MapReduce job and task trackers spawn map or reduce tasks for the job, and send heartbeat messages and job status updates to the job tracker. It is relevant here to note that Hadoop allocates resources in a node as *task slots*. Each node may have multiple task slots depending on the number of CPU cores and physical memory available for processing. Typically 3-4 slots can be configured on a node with 4 CPU cores.

2.5.2 Request Handler

Penny consists of Penny agents that are inserted between Pig script states. These agents in turn are implemented as user defined functions that can exchange messages with other agents and a Penny coordinator. Our changes involve creating a verifying function as a Penny tool that creates a SHA-256 digest and sends the digest back to the coordinator in the trusted tier. We modified the Penny infrastructure to allow creation of multiple coordinators, so that different replicas can reply back to different coordinators.

2.5.3 Execution Handler

We implement the resource manager by creating a new task scheduler that extends the `TaskScheduler` class in Hadoop. Hadoop allows creation of multiple *job queues* to which jobs can be submitted. In ClusterBFT each replica of a job can be submitted to one queue. In order to tolerate faulty nodes, we also need to ensure that tasks from more than one replica of a job are not scheduled on a same node at any point of time. Such a collocation could result in one faulty node modifying the outcome of more than one replica and thus violating safety. Note that this does not prevent us from collocating tasks from different jobs on the same node. We added data structures to the `JobInProgress` class that will keep track of replica information to prevent this during task scheduling. We also added a new alphanumeric parameter `sub.graph.id` to the `JobConf` class. `sub.graph.id` corresponds to *sid* in Section 2.4.1 and is set during job initiation. All replicas of a single job must have the same `sub.graph.id`. `JobTracker` itself works without any modifications as our execution tracker.

2.5.4 Ensuring Determinism

The data parallelism leveraged by MapReduce may naturally lead to non-determinism, which can be observed through differing digest values across replicas even without

faulty processes. For example in order to calculate an average, instead of finding the sum of all values of a key and dividing it by the number of values, users may decide to maintain a moving average, causing final outputs to differ (in the least few significant bits of precision). Our current prototype works around this issue by ensuring that the user programs deal with only integer values or truncate the last few decimal points before performing arithmetic operations. For a more general solution we intend to address this issue in future work by ordering the intermediate mapper output based on mapper ids.

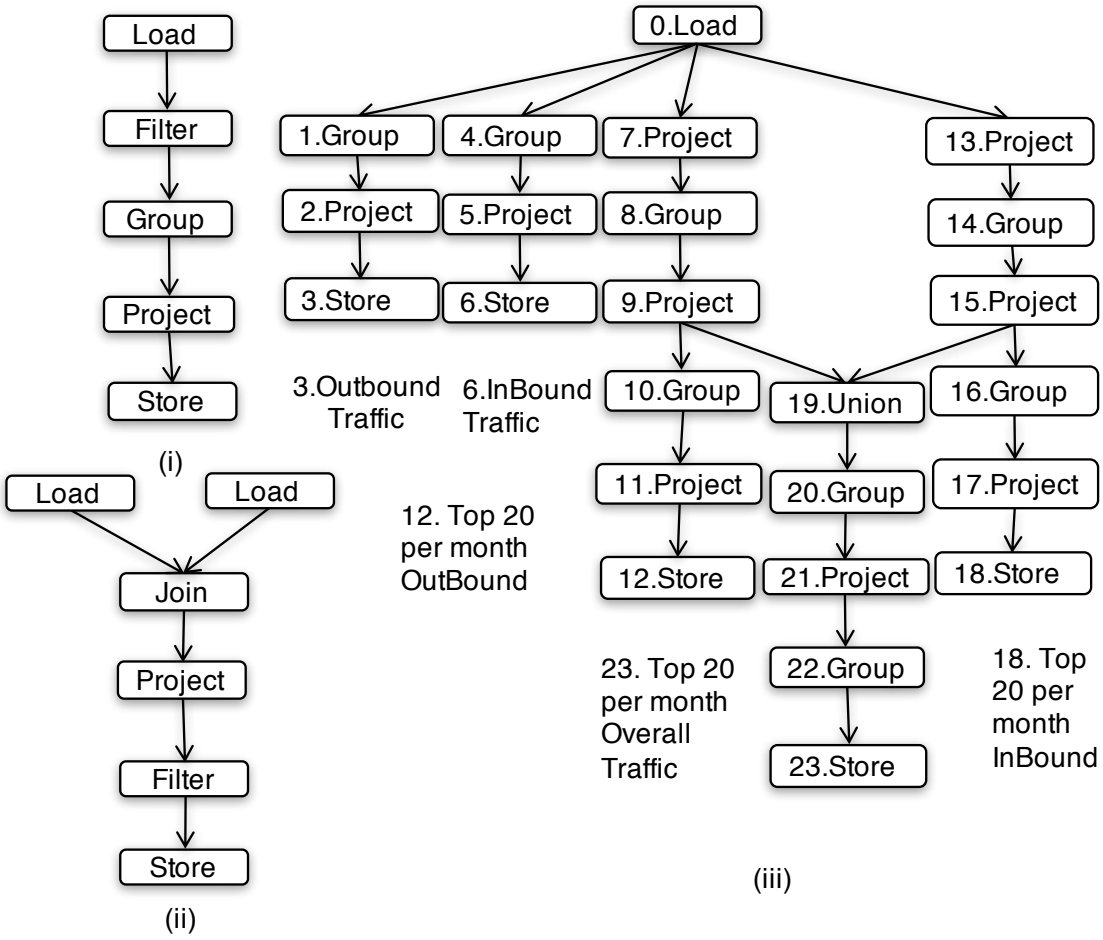


Fig. 2.5.: Data-flow graph for (i) Twitter Follower Analysis (ii) Twitter Two Hop Analysis, (iii) Air Traffic Analysis

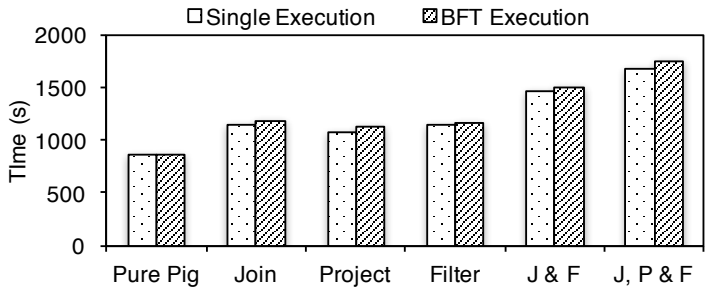


Fig. 2.6.: Digest computation overhead for Twitter Two Hop Analysis

2.6 Evaluation

To assess the benefits of our approach, we evaluate (a) overhead incurred by ClusterBFT, (b) gains of ClusterBFT under different replication degrees in the presence of failures (c) effectiveness of fault isolation algorithm and (d) system performance for higher approximation accuracy. **Setup.** For evaluations in Section 2.6.1 and 2.6.2, we use planet-lab based Vicci [22] as our testbed. Machines are 12-core Intel Xeon servers with 48GB RAM virtualized using Linux containers. Our untrusted tier consists of 32 nodes and our trusted tier consists of 2 nodes. We use Amazon EC2 for the evaluation in Section 2.6.4 with 8 nodes in the untrusted tier and 4 nodes in the trusted tier.

2.6.1 Verification Overhead: Twitter Data Analysis

First we measure the overhead involved in computing digests required for verification. For this we use the Twitter data-set from [23] and compute *SHA-256* digests at different points for two Pig scripts. The data-set consists of two columns, *user-id* and *follower-id* represented as numeric values. We run two Pig scripts outlined in [24]. The first script (Twitter Follower Analysis) counts the number of *followers* for each user. It loads the data, filters out empty records, groups the record by *user-id*, calculates the counts and saves the *user-id* and respective counts. The second script (Twitter Two Hop Analysis) lists pairs of users that are are two hops away from one

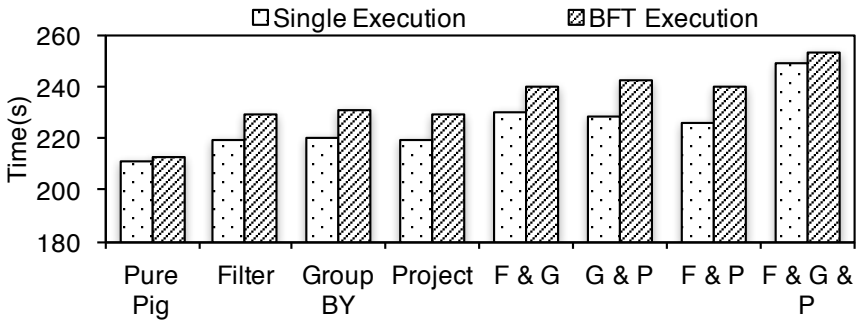


Fig. 2.7.: Latency of running Twitter Follower Analysis

another. This job does a self-join that matches one user with all its *follower's followers*. The data-flow graphs for these two scripts are presented in Figure 2.5 (i) and (ii) respectively. Figures 2.7 and 2.6 shows the total time taken for job completion when digests are computed at different points of the respective jobs. In both graphs, *Single Execution* shows the time taken by a single replica of the script and *BFT Execution* shows the time taken by 4 replicas of the script to execute. *BFT Execution* also includes the overhead of matching $f + 1$ digests generated by the replicas. *Pure Pig* shows the baseline run with no verification points or replication. When digests are computed at multiple points in the data-flow graph, it is abbreviated using the first letter of the verification point. When digests are computed at multiple points in the data-flow graph, it is abbreviated using the first letter of the verification point. Figure 2.7 show a minimal overhead of 8% and worst case of 9%, 14% and 19% overhead with 1, 2 and 3 verification points respectively.

Table 2.3.: ClusterBFT in the presence of Byzantine failures

	r =2		r = 3, case 1		r = 3, case 2		r = 4	
Measure	C	P	C	P	C	P	C	P
Latency (s)	1.6×	2.1×	1.1×	1.1×	1.6×	2.1×	1.1×	1.1×
CPU time spent (ms)	3.5×	4.1×	3.1×	3.1×	4.5×	6.2×	4.2×	4.2×
File read (Bytes)	3.6×	4×	2.6×	3×	4.7×	6×	3.6×	4×
File write(Bytes)	3.4×	4×	2.4×	3×	4.7×	6×	3.4×	4×
HDFS write (Bytes)	2×	4×	2×	3×	2×	6×	3×	4×

2.6.2 Performance under Failures: IRTA Airline Traffic Analysis

Next we look at ClusterBFT's performance in the presence of node failures. The input data-set for this evaluation is a 1.3GB subset of airline data-set provided by RITA [25]. We run a multi-store query outlined in [24] that finds the top 20 airports with respect to incoming flights, outgoing flights, and overall. The data-flow graph for this script is shown in Figure 2.5 (iii). The evaluation is set up for $f = 1$ and we show the benefits of ClusterBFT under various replication degrees with 2 verification points. We compare ClusterBFT (C in Table 2.3) with modified version of Pig which verifies digest of the final output only and not anywhere else in the data-flow graph (P in Table 2.3). The results are shown in terms of a multiplier over a single run of standard Pig without replication or digest computation. For both executions (C and P), one node was set up to always produce commission failures resulting in an incorrect digest. Also for $r = 3$, we took two measurements. The first measurement (case 1) shows results when all computations got done within the verifier timeout value. The second measurement (case 2) shows one correct replica not responding within the verifier timeout causing the script to be scheduled again with higher timeout value. Results show that latency decreases by 23% ($r = 2, r = 3$ case 2) for test runs that require rescheduling. For runs that do not require rescheduling, our latency is on par with running multiple replicas, and show up to 14% reduced overhead.

2.6.3 Effectiveness of Fault Isolation: Simulation

Next we evaluate the fault analyzer algorithm outlined in Figure 2. We wrote a Java-based simulator that mimics resource allocation in a 250 node Hadoop cluster. Each node is given 3 slots on which tasks can be scheduled. We consider jobs as falling under three categories: *large* (requiring 20 to 30 slots), *medium* (10 to 15 slots) and *small* (3 to 5 slots). The exact number of slots is determined uniformly at random. Each job is also associated with a unit of time as length. We studied the algorithm under various ratios of *small*, *medium* and *large* jobs as well as various length for jobs.

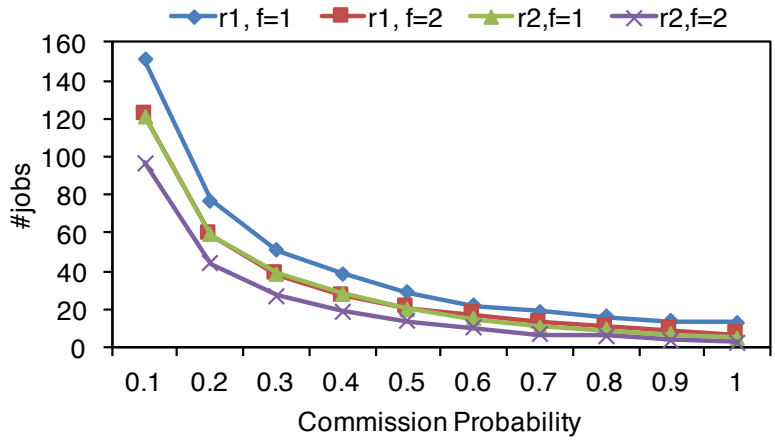


Fig. 2.8.: Number of jobs required to identify disjoint set of faults

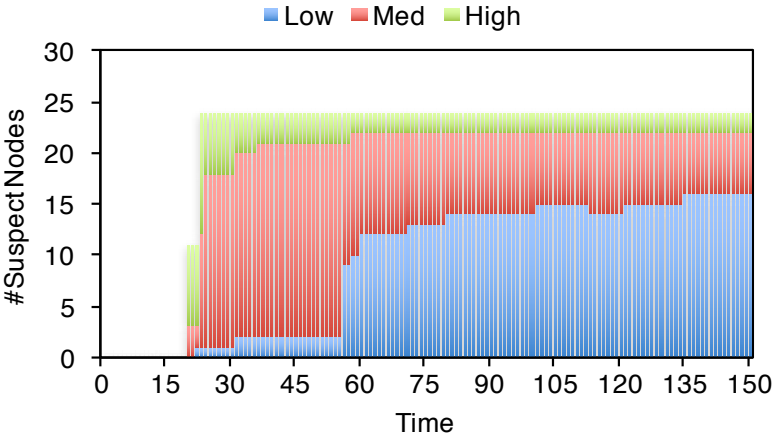


Fig. 2.9.: Suspicion level changes over time

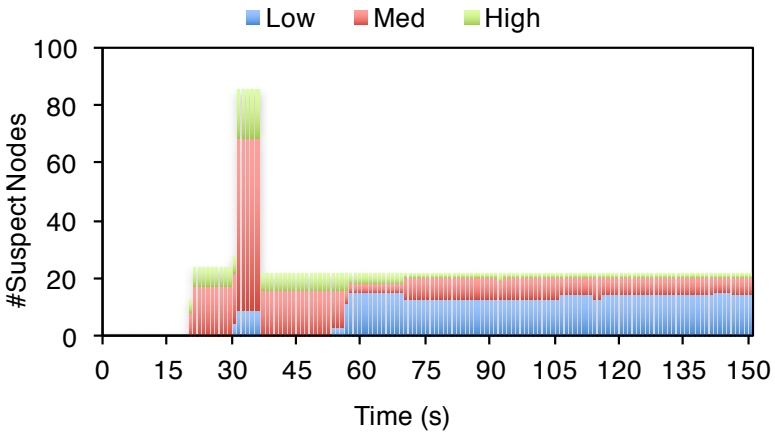


Fig. 2.10.: Suspicion level spike as a result of multiple large clusters with faulty nodes

We present a subset of our results here. Figure 2.8 shows the average number of jobs that got completed when the number of disjoint faulty sets (D) becomes equal to f (Figure 2 line 12). This point is important because the number of suspicious nodes will not increase after this point. We show measurements for two ratios of job sizes and two values of f. Job size ratio $r1$ indicates $|large| : |medium| : |small| = 6 : 3 : 1$ and r2 indicates $2 : 2 : 1$. For $f = 1$, we used 4 replicas and $f = 2$, we used 7 replicas. The abscissa shows the probability with which a faulty node produces a commission failure. This result shows that if a node produces commission faults with very high probability, then by the time 10 jobs complete execution, we can isolate the fault to a much smaller subset. If a node produces commission faults with probability of .6 or more, less than 20 jobs are required to isolate the fault. The size of these subsets indicate the number of suspicious nodes, and this is explored in Figure 2.9 and Figure 2.10. In order to understand the number of nodes suspected by the algorithm and the suspicion level (s) of these nodes, we group suspicion level into four categories: no suspicion, *Low* (with $0 < s <= 0.33$), *Med* ($0.33 < s \leq 0.66$) and *High* ($0.66 < s \leq 1$). The goal of the algorithm is to narrow the suspicion down to fewer nodes, or in other words, we should have less nodes with high value for s. Figure 2.9 shows how s changes with time. The initial values ($Time < 15$) indicates that no job has so far showed a commission fault. After this point we see that the number of nodes with $s > 0$ increases. It is also worthwhile to note that at around $Time = 25$, $|D|$ becomes equal to f and the number of nodes with $s > 0$ does not increase further. The graph clearly shows that nodes start with *High* and *Med* suspicion levels, but over time, the suspicion levels of faulty nodes remain *High*, and of others are reduced. In fact, in these trials, by $Time = 50$, only the real faulty nodes were left in the *High* suspicion category. In Figure 2.10, we show occasional spikes in the number of suspicious nodes that we observed in some of the runs. This happens before $|D|$ becomes equal to f. This is because it may so happen that two replicas of *large* jobs show commission fault and all nodes in them gets a non zero

value for s. But within a few more runs the algorithm prunes the suspicion list and increasingly suspects the real faulty nodes as can be seen when $Time > 35$.

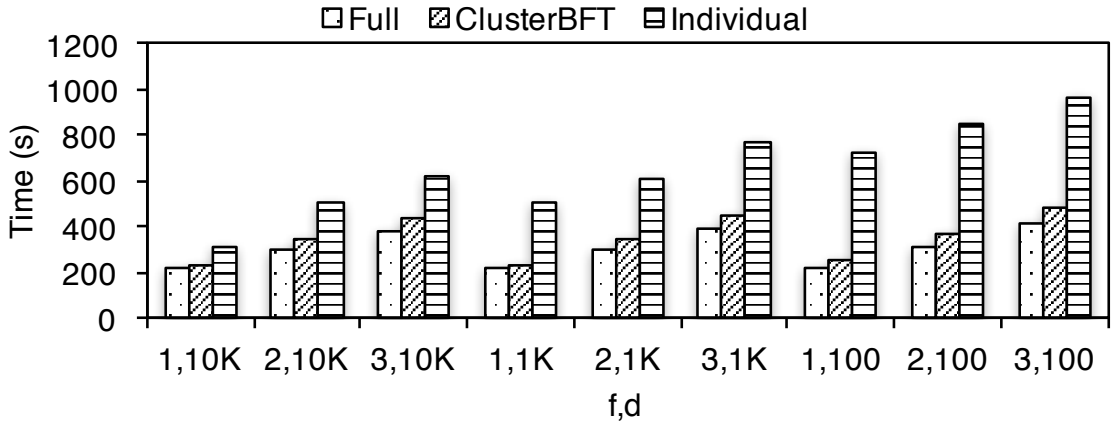


Fig. 2.11.: Computing average weather temperatures

2.6.4 Approximation Accuracy: Weather Average Temperature

Here we test how ClusterBFT performs if we increase the approximation accuracy from the default, one digest at one verification point, to multiple digests at each verification points. For this experiment we move away from the assumption of implicit trust within the trusted tier and instantiate $3f + 1$ replicas of the request handler. We use the BFT-SMaRT [26] library for achieving Byzantine fault tolerance within these request handler replicas. Input data for this experiment is a 640MB subset of the Daily Surface Summary of Day weather data [27]. The script involves finding average temperate over multiple years for each weather station followed by counting the number of stations with the same average. We take measurements for different values of f and change the number of lines d for which a digest is created. Figure 2.11 shows the results. In the figure, *Full* refers to script execution with digest computed and verified only for the output. *ClusterBFT* refers to using ClusterBFT with 2 verification points and *Individual* refers to digest computed for each vertex of the

data-flow graph. Results show that latency overhead of ClusterBFT is within 10-18% of full replication even with increasing approximation accuracy.

3. CONFIDENTIALITY IN BATCH PROCESSING SYSTEMS

This chapter presents our solution for performing confidential batch data analytics (published in [28, 29]).

3.1 Overview

The cloud computing model has evolved as a cost-efficient data analysis platform for corporations, governments and other institutions. A rich set of tools like MapReduce [6], Dryad [30], or Spark [13] — just to name a few — allow developers to execute data analysis jobs in the cloud easily. However, in order to take advantage of these possibilities, developers are faced with the decision of moving sensitive data to the cloud, and placing trust on infrastructure providers. Even with a trusted provider, malicious users and programs and defective software can leak data.

3.1.1 Computing on Encrypted Data

Several existing cryptographic systems ("cryptosystems") allow meaningful operations to be performed directly on encrypted data. Advances occurring regularly in *fully homomorphic encryption* (FHE) schemes (e.g. [31]) further increase the scope and performance of computations on encrypted data. These cryptosystems present an opportunity to maintain sensitive data only in an encrypted form in the cloud while still enabling meaningful data analysis. Even within an enterprise, maintaining data in an encrypted format helps prevent insider attacks and accidental leaks.

Unfortunately, such advancements in cryptography have not translated into programmer friendly frameworks for big data analysis with acceptable performance. Cur-

rently, for a big data programmer to *efficiently* put existing homomorphic schemes to work she will have to explicitly make use of corresponding cryptographic primitives in her data analysis jobs. Consider a simple program that finds the total marks obtained by students: the programmer is burdened by the task of identifying an encryption scheme suitable to represent such an operation (*Paillier* [32] in this example with addition) and express the operation (summing) in cryptographic terms (multiplication of encrypted operands followed by a modulo division by the square of the public key).

To make this task even more difficult for the programmer, big data analysis jobs are typically expressed in *domain-specific* security-agnostic high-level data flow languages, like Pig Latin [9] or FlumeJava [33], which are commonly compiled to *sequences* of several MapReduce tasks [6]. This makes the explicit use of cryptographic primitives even harder. Recent approaches to automatically leveraging partially homomorphic encryption (PHE) — e.g., addition of values encrypted with Paillier cryptosystem via multiplication — for secure cloud-based computing target different application scenarios or work-loads. For instance, the MySQL-based CryptDB [34] supports SQL queries on encrypted data yet does not enable parallelization through MapReduce; MrCrypt [35] only supports individual MapReduce tasks.

3.1.2 Approach

In this chapter we present a program analysis and transformation scheme for the Pig Latin high-level big data analysis language which enable the efficient execution of corresponding scripts by exploiting cloud resources but without exposing data in the clear. We implemented this program analysis and transformation inside a secure Pig Latin runtime (SPR). More specifically, we extend the scope of encryption-enabled big data analysis based on the following insights:

Extended program perspective: By *analyzing entire data flow programs*, SPR can identify many opportunities for operating in encrypted mode. For example, SPR can identify operations in Pig Latin scripts that are inter-dependent with respect to

encryption, or inversely, independent of each other. More precisely, when applying two (or more) operations to the same data item, many times the second operation does not use any *side-effect* of the former, but operates on the original field value. Thus, *multiple encryptions* of a same field can support different operations by carefully handling relationships between encryptions.

Extended system perspective: By considering the possibility of performing subcomputations on the client side, SPR can still exploit cloud resources rather than giving up and forcing users to run entire data flow programs in their limited local infrastructure, or defaulting to FHE (and then aborting [35]) when PHE does not suffice. For example, several programs in the PigMix (I+II) benchmarks [36] end up averaging over the values of a given attribute for several records after performing some initial filtering and computations. While the summation underlying averaging can be performed in the cloud via an *additive homomorphic encryption* (AHE) scheme, the subsequent division can be performed on the client side.

Considering that the amount of data continuously decreases as computation advances in most analysis jobs, it makes sense to *compute as much as possible in the cloud*.

3.1.3 Contributions

The contributions of this chapter are as follows. After presenting background information on homomorphic encryption and Pig Latin, we

1. propose an execution model for executing Pig Latin scripts in the cloud without sacrificing confidentiality of data.
2. outline a novel data flow analysis and transformation technique for Pig Latin that distinguishes between operations with side-effects (e.g., whose results are used to create new intermediate data) and without (e.g., filters). The results are semantically equivalent programs executable by SPR that maximize the amount of computations done on encrypted data in the cloud.

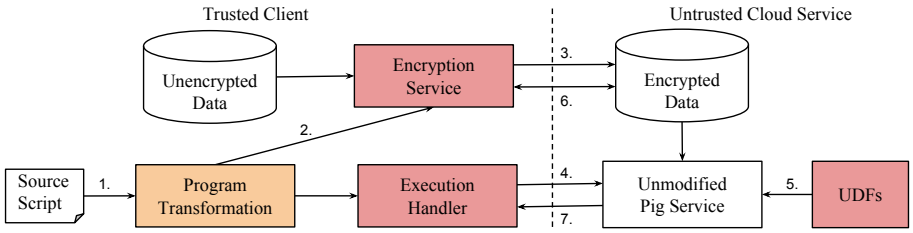


Fig. 3.1.: Execution Model of SPR. Shading indicates components introduced by SPR.

3. present initial evaluation results for an implementation of our solution based on the runtime of Pig Latin scripts obtained from the open-source Apache Pig [8] PigMix benchmarks.

The remainder of this chapter is organized as follows. Section 3.2 presents background information on homomorphic encryption and Pig Latin. Section 3.3 outlines the design of our solution. Section 3.4 presents our program analysis and transformation. Section 3.5 outlines the implementation of SPR. Section 3.6 presents empirical evaluation. Section 3.7 discusses limitations. Section 5.2 contrasts with related work.

3.2 Background

This section presents background information on homomorphic encryption and our target language Pig Latin.

3.2.1 Homomorphic Encryption

A cryptosystem is said to be *homomorphic* (with respect to certain operations) if it allows computations (consisting in such operations) on encrypted data. If $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively, then a cryptosystem is said to be homomorphic with respect to addition if $\exists \psi$ s.t.

$$D(E(x_1)\psi E(x_2)) = x_1 + x_2$$

Dually a cryptosystem is said to provide *additive homomorphic encryption* (AHE). Similarly a cryptosystem is said to be homomorphic with respect to multiplication or support *multiplicative homomorphic encryption* (MHE) if $\exists\chi$ s.t.

$$D(E(x_1)\chi E(x_2)) = x_1 \times x_2$$

Other homomorphisms are with respect to operators such as "$\leq$" and "$\geq$" (*order-preserving encryption* – OPE) or equality comparison "=" (*deterministic* encryption – DET). *Randomized* (RAN) encryption does not support any operators, and is intuitively, the most desirable form of encryption because it does not allow an attacker to learn anything.

3.2.2 Pig/Pig Latin

Apache Pig [8] is a data analysis platform which includes the Pig runtime system for the high-level data flow language Pig Latin [9]. Pig Latin expresses data analysis jobs as sequences of data transformations, and is compiled to MapReduce [6] tasks by Pig. The MapReduce tasks are then executed by Hadoop [20] and output data is presented as a folder in HDFS [37]. Pig allows data analysts to query big data without the complexity of writing MapReduce programs. Also, Pig does not require a fixed schema to operate, allowing seamless interoperability with other applications in the enterprise ecosystem. These desirable properties of Pig Latin as well as its wide adoption[1] prompted us to select it as the data flow language for SPR.

We give a short overview of Pig Latin here and refer the reader to [9] for more details.

Types. Pig Latin includes *simple types* and *complex types*. The former include signed 32-bit and 64-bit integers (`int` and `long` respectively), 32-bit and 64-bit floating point

[1]According to IBM [38], "*Yahoo estimates that between 40% and 60% of its Hadoop workloads are generated from Pig [...] scripts. With 100,000 CPUs at Yahoo and roughly 50% running Hadoop, that's a lot[...]*".

values (`float`, `double`), arrays of characters and bytes (`chararray`, `bytearray`), and `boolean`s. Pig Latin also pre-defines certain values for these types (e.g., `null`).

Complex types include the following:

- `bag`s `{...}` are collections of `tuple`s.
- `tuple`s `(...)` are ordered sets of fields.
- `map`s `[...]` are sets of key-value pairs *key*`#`*value*.

Furthermore, a *field* is a data item, which can be a `bag`, `tuple`, or `map`. Pig Latin statements work with *relations*; a relation is simply a (outermost) `bag` of `tuple`s. Relations are referred to by named variables called *aliases*. Pig Latin supports assignments to variables.

Operators and expressions. Relations are also created by applying operators to other relations. The main relational operators include:

`JOIN` This same operator is used with various parameters to distinguish between inner and outer joins. The syntax closely adheres to the SQL standard.

`GROUP` Elements of several relations can be grouped according to various criteria. Note that `GROUP` creates a nested set of output `tuple`s while `JOIN` creates a flat set of output `tuple`s.

`FOREACH...GENERATE` Generates data transformations based on columns of data.

`FILTER` This operator is used with `tuple`s or rows of data, rather than with columns of data as `FOREACH...GENERATE`.

Operators also include arithmetic operators (e.g.,`+, -, \, *`), comparisons, casts, and `STORE` and `LOAD` operators.

Pig Latin is an expression-oriented language. Expressions are written in conventional mathematical infix notation, and can include operators and functions described next.

Functions. Pig Latin includes *built-in* and *user-defined* functions. The former include several different categories:

- *Eval* functions operate on `tuple`s. Examples include `AVG`, `COUNT`, `CONCAT`, `SUM`, `TOKENIZE`.
- *Math* functions are self-explanatory. Examples include `ABS` or `COS`.
- *String* functions operate on character strings. Examples include `SUBSTRING` or `TRIM`.

User-defined functions (UDFs) in Pig Latin are classified as: (1) built-in UDFs, or (2) custom UDFs. The former are pre-defined functions that are delivered with the language toolchain. Examples include "composed" functions like `SUM`. In this chapter we focus on the former kind of UDFs due to the complexity involved in analyzing the latter.

Example. To understand the constructs of Pig Latin, we present a simple word count example in Listing 3.1. In this example, a relation `input_lines` is first generated with elements called `line` of type `chararray` read from an input `input_file`. Next, the script breaks these lines into tokens, representing individual words. Then, occurrences of a same word are `GROUP`ed, and their respective number of occurrences `COUNT`ed. Finally the data in this table is `STORE`d in a file `output_file`.

```
1 input_lines = LOAD 'input_file' AS (line:chararray);
2 words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS
    word;
3 word_groups = GROUP words BY word;
4 word_count = FOREACH word_groups GENERATE group, COUNT(words);
5 STORE word_count INTO 'output_file';
```

Listing 3.1: Source Pig Latin script S_1

3.3 Execution Model

SPR is a runtime for Pig Latin that supports cloud-based execution of scripts written in the original Pig Latin language in a confidentiality-preserving manner. The adversary in our model can have full control of the cloud infrastructure. This means the adversary can see all data stored in the cloud and the scripts that operate on data. In order to preserve confidentiality in the presence of such an adversary only encrypted data is maintained in the cloud and SPR operates on this encrypted data. However, SPR does not address integrity and availability issues. (Corresponding solutions are described in our previous work [4] which inversely, however, does not address confidentiality.)

Figure 3.1 presents an overview of the execution model of SPR. Script execution proceeds by the following steps:

1. Program transformation. Script execution starts when a user submits a Pig Latin script operating on unencrypted data (source script). SPR analyzes the script to identify the *required encryption schemes* under which the input data should be encrypted. For the transformed script to operate on encrypted data, operators in the source script are replaced with calls to secure UDFs that perform the corresponding operations on encrypted data. E.g., an addition $a + b$ in the source script will be replaced by $a \times b \mod n^2$, with n the public key used for encrypting a and b. Constants are also replaced with their encrypted values to generate a target script that executes entirely on encrypted data. Details of this transformation yielding an *encryption-enabled* script are presented in Section 3.4.

2. Infer encryption schemes. Using the input file names used in the source script, the encryption service checks what parts of the input data are already encrypted and stored in the cloud. Some parts of the input data might already be encrypted under multiple encryption schemes (to support multiple operations) and other parts might not be available in the cloud at all. The encryption service maintains an *input data encryption schema* which keeps track of this mapping between plain text input data

and encrypted data available in the cloud. Based on the *input data encryption schema* and the *required encryption schemes* inferred in the previous step, the encryption service identifies the encryption schemes missing from the cloud.

3. Encrypt and send. Once the *required encryption schemes* that are missing from the cloud is identified, the encryption service loads the unencrypted data from local storage, encrypts it appropriately and sends it to the cloud storage. The encryption service makes use of several encryption schemes each implemented using different cryptosystems. Implementation details of these encryption schemes are presented in 3.5. Each of these encryption schemes has its own characteristics. The first scheme is randomized (RAN) encryption which does not support any operators, and is intuitively, the most secure encryption scheme. The next scheme is the deterministic (DET) encryption scheme which allows equality comparisons over encrypted data. Order-preserving encryption (OPE) scheme allows order comparisons using the order-preserving symmetric encryption [39]. Lastly, additive homomorphic encryption (AHE) allows additions over encrypted data, and multiplicative homomorphic encryption (MHE) allows us to perform multiplications over encrypted data.

4. Execute transformed script. When all required encrypted data are loaded in the cloud, the execution handler issues a request to start executing the target script.

5. UDFs. SPR defines a set of pre-defined UDFs that handle cryptographic operations. Such UDFs are used to perform operations like additions and multiplications over encrypted data. The target script calls these UDFs using standard Pig Latin syntax as part of the script execution process.

6. Re-encryption. During the target script execution, intermediate data may be generated as operations are performed. The encryption scheme of that data depends on the last operation performed on that data. For example, after an addition operation, the resulting sum is already encrypted under AHE. If that intermediate data is subsequently involved in an operation that requires an encryption scheme other than the one it is encrypted under (for example multiplying the sum with another

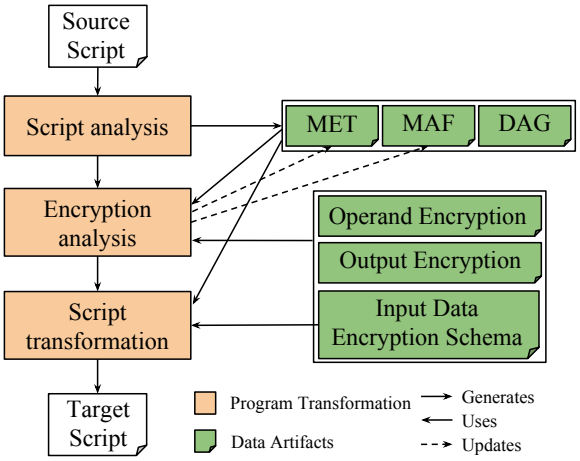


Fig. 3.2.: Program transformation components and artifacts

value requires MHE), the operation cannot be performed. (The ability to apply additions and multiplications in sequence is viewed as defining characteristic of FHE.) SPR handles this situation by re-encrypting the intermediate data. Specifically, the intermediate data is sent to the client where it can be securely decrypted and then encrypted under the required encryption scheme (for example the sum is re-encrypted under MHE), before sent back to the cloud. Once the re-encryption is complete, the execution of target script can proceed.

7. Results. Once the job is complete, the encrypted results are sent to the client where they can be decrypted.

3.4 Program Analysis and Transformation

In this section we give a high level overview of how Pig Latin scripts are analyzed by SPR and transformed to enable execution over encrypted input data.

3.4.1 Running Example

We use the Pig Latin script shown in Listing 3.2 as a running example to explain the analysis and subsequent transformation process. This script is representative of

the most commonly used relational operations in Pig Latin and allows us to explain key features about SPR. The script loads two input files: `input1` with two fields and `input2` with a single field. The script then filters out all rows from `input1` which are less than or equal to 10 (Line 3). Lines 4 and 5 group `input1` by the first field and find the sum of the second field for each group. Line 6 joins the sum per group with the second input file `input2` to produce the final result which is stored into an output file `out` (Line 7).

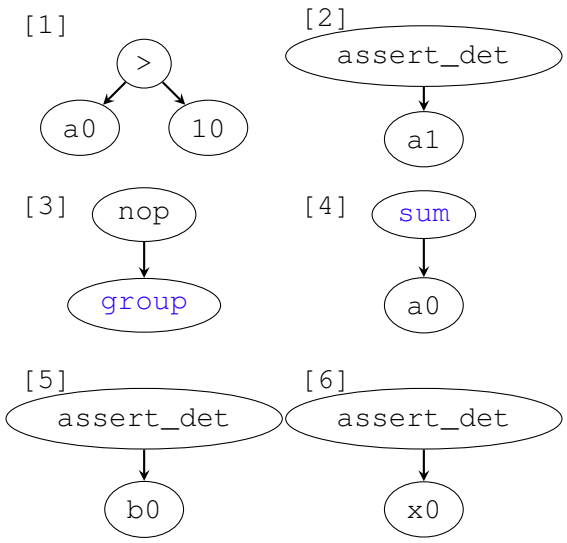
```
1 A = LOAD 'input1' AS (a0, a1);
2 B = LOAD 'input2' AS (x0);
3 C = FILTER A BY a0 > 10;
4 D = GROUP C BY a1 ;
5 E = FOREACH D GENERATE group AS b0, SUM(C.a0) AS b1;
6 F = JOIN E BY b0, B BY x0;
7 STORE F into 'out';
```

Listing 3.2: Source Pig Latin script S_1

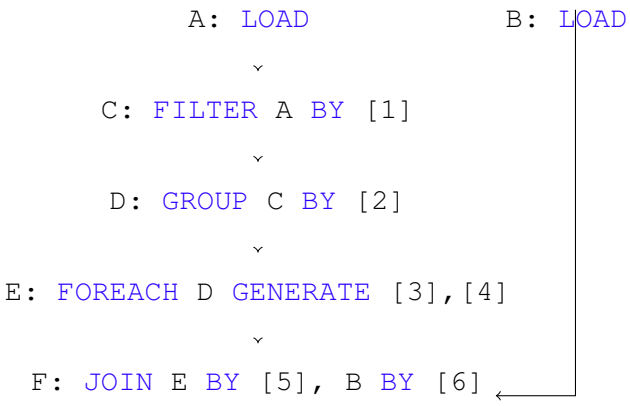
3.4.2 Definitions

In order to describe our program analysis and transformation we first introduce the following definitions.

Map of expression trees (MET). All the expressions that are part of the source script are represented as trees and added to the *map of expression trees* (MET) as values. The keys of the MET are simple literals used to access these expression trees. Figure 3.3a shows the MET for the Pig Latin script in Listing 3.2. Keys of the MET are shown in square brackets. Note that operands that are not part of any expression are surrounded by a *nop* (no operation) vertex and operands that are part of a group by or join operation are surrounded by an `assert_det` (assert deterministic) vertex. The `assert_det` vertex can be seen as a simple operator that requires its operand to be present in DET encryption.



(a) Map of expression trees



(b) Data flow graph

Fig. 3.3.: MET and DFG for Pig Latin script in Listing 2. Keys of MET are shown in square brackets.

Data flow graph. We represent a Pig Latin script S using a data flow graph (DFG). More precisely, $S = \{V, E\}$, where V is a set of vertices and E is a set of ordered pairs of vertices. Each $v \in V$ represents a relation in S. Each $e \in E$ represents a data flow dependency between two vertices in V. Note that it follows from the nature of the Pig Latin language that DFG is acyclic. For describing our analysis, we consider a vertex $v \in V$ representing the Pig Latin relation R to have two parts: (a) the

Pig Latin statement representing R with all expressions replaced by their keys in the MET and (b) the fields in the schema of R.

Programmatically a data flow graph G exposes the following interface

- Iterator getIter() Returns an iterator representing the list of relations in G in topologically sorted order.

Annotated field (AF). We represent each field in the schema of each relation by an annotated field abstraction. In other words there will be one AF for every $\langle relation, field \rangle$ pair. The intution behind using AFs instead of actual field names is that in the transformed program one field in the source script may be loaded as multiple fields, each under a different encryption scheme. Each AF consists of the following three annotations.

- parent: AFs track their lineage using this parent field. The parent of an annotated field could be another annotated field or an expression tree in the MET. In cases where an AF represents a field which is newly generated as part of a Pig Latin relational operation (`GROUP ..BY`, `FOREACH` etc.), the parent will be the expression tree used to generate it. Otherwise, the parent of an AF will be the corresponding AF in the vertex in the DAG that comes before it.

- avail: This annotation represents the set of encryption schemes available for the field being represented. At the begining of the analysis this property will be empty. This will be filled in as part of the program analysis.

- req: This represents the encryption scheme that is required for this field by our execution engine for encryption-enabled execution.

Map of annotated fields (MAF). We capture the mapping between $\langle relation, field \rangle$ pairs and corresponding AFs using *map of annotated fields* (MAF). The MAF for the Pig Latin script in Listing 3.2 is shown in Table 3.1. Note that annotated fields are unique within a Pig Latin script which allows us to implement the MAF as a bidirectional map allowing lookups based on keys as well as values.

Table 3.1.: Map of annotated fields for Pig Latin script in Listing 3.2. †Keys are pairs of $\langle relation, field \rangle$ and ‡values are corresponding annotated fields. NE represents no encryption.

Key†	Value‡	After analysis
A,a0	f0<null,NE,NE>	f0<null,{ALL},OPE>
A,a1	f1<null,NE,NE>	f1<null,{ALL},NE>
B,x0	f2<null,NE,NE>	f2<null,{ALL},DET>
C,a0	f3<f0,NE,NE>	f3<f0,{ALL},NE>
C,a1	f4<f1,NE,NE>	f4<a1,{ALL},DET>
D,gr	f5<[2],NE,NE>	f5<[2],{DET},NE>
D,a0	f6<f3,NE,NE>	f6<f3,{ALL},AHE>
D,a1	f7<f4,NE,NE>	f7<f4,{ALL},NE>
E,b0	f8<[3],NE,NE>	f8<[3],{DET},DET>
E,b1	f9<[4],NE,NE>	f9<[4],{AHE},NE>
F,b0	f10<f4,NE,NE>	f10<f4,{ALL},NE>
F,b1	f11<f3,NE,NE>	f11<f3,{AHE},NE>
F,x0	f12<f3,NE,NE>	f12<f3,{ALL},NE>

Algorithm 3 Determining available encryption scheme for fields in MAF fs for a DFG G

```
1: met                                                    ▷ MET for DFG G
2: procedure AVAILABLEENC(G, fs)
3:     iter ← G.getIter();
4:     while r ← iter.next() do
5:         anFields ← GETAFS(fs, r)
6:         for each af ∈ anFields do
7:             FINDAVAIL(af)
8:     end while
9: end procedure
10: function FINDAVAIL(af)                                ▷ af is Annotated Field
11:     if af.parent is null then
12:         af.avail ← ENCSERVICE(met.getkey(af))
13:     else if af.parent ∈ met.keys() then
14:         exprTree ← met.get(af.parent)
15:         af.avail ← OUTENC(exprTree.root.operator)
16:     else
17:         af.avail ← af.parent.avail
18: end function
```

Table 3.2.: Description of functions used in program analysis and transformation

Function name	Description
getAFs(maf, r)	Returns list of annotated fields which contain the relation r as a part of their key in maf
outEnc(op)	Returns the encryption scheme in which the result will be after performing the operation op
inEnc(op)	Returns encryption schemes in which the operands of operator op much be specified
addToList($list, elem$)	Adds $elem$ to the list of values represented by $list$
replaceAF(af, fld)	Replaces all occurrences of the annotated field af in expressions and relations with the field fld
insertVertex(v, bv)	Inserts new vertex v before vertex bv in met. The parent of v is set to be the parent of bv and the parent of bv is set to be v

3.4.3 Analysis

Using the programming abstractions defined above, we describe the program analysis and transformation that we perform. Before the analysis, as part of initialization, all operators and UDFs to be used in the script are pre-registered with the encryption scheme required for operands and the encryption scheme of output generated. Some Pig Latin relational operators also require fields to be in specific encryption schemes. For example, the field or expression which is the grouped field of `GROUP BY`, require the field or expression to be available in DET encryption. Further the encryption scheme of the new "group" field generated by the `GROUP BY` operator will also be available in DET encryption. For example, the `assert_det` function that we introduced will be registered with required encryption scheme as DET and output encryption scheme also as DET. To keep the description simple, we encapsulate this static information as two function calls in our algorithm: outEnc($oper$) which returns the output encryption scheme of the operator $oper$ and inEnc($oper$) which returns the operand encryption scheme required for $oper$. These functions are summarized in Table 3.2.

The available encryption scheme for each AF is identified by observing the lineage information available through the parent field of AFs and metadata about the encrypted input file. Details of available encryption scheme analysis is presented by Algorithm 3. The available encryption schemes for AFs part of the `LOAD` operation are set based on metadata about the encrypted file. For other than `LOAD` operators, the available encryption schemes for AFs depend on their lineage. If the AF is derived by an expression, available schemes for the AF are determined by the deriving expression. For example, the available encryption scheme for the AF representing field `b1` in `B = FOREACH A GENERATE a0+a1 as b1;` is determined by the expression `a0 + a1` or more precisely by the operator `+`. If the AF is not explicitly derived, but is carried forward from a parent relation, then the AF simply inherits the available encryption schemes of the parent AF. For example, the available encryption schemes for the AF representing field `a0` in `B = FILTER A BY a0 < 10;` is same as the encryption schemes available for the parent AF representing field `a0` in relation `A`.

Next we describe how the required encryption field is populated in each AF. As part of our initialization, once the MAF is generated, we replace each field in the MET by the AF representing that field. Once this is done, the required encryption for each AF can be identified by iterating over all leaf vertices of all expression trees in the MET. The required encryption scheme for each leaf vertex is the same as inEnc(*parent_operator*), where *parent_operator* is the operator for which the leaf vertex is the operand. This procedure is thus straightforward and we do not present this as a separate algorithm.

3.4.4 Transformation

Next we describe how a Pig Latin script, represented as $S = \{V, E\}$, is transformed into the encryption-enabled target script. We describe the transformation process in multiple sections. In each section we give an overview of why we perform a specific transformation and then describe the transformation process itself.

```
A = LOAD 'enc_input1' AS (a0_ope, a0_ah, a1_det);
B = LOAD 'enc_input2' AS (x0_det);
C = FILTER A BY OPE_GREATER(a0_ope ,
  '0xD0004D3D841327F2CCE7133ABE1EFC14');
D = GROUP C BY a1_det ;
E = FOREACH D GENERATE group AS b0, SUM(B.a0_ah) AS b1;
F = JOIN E BY b0, B BY x0_det;
STORE F into 'out';
```

Listing 3.3: Transformed Pig Latin script

Algorithm 4 Program transformation

```
1: S                                                    ▷ Pig Latin script S
2: met                                                  ▷ MET
3: file                                                 ▷ input file used in LOAD
4: procedure TRANSFORMMAIN(S)
5:     for each input_file ∈ S do
6:         TRANSFORM(input_file)
7: end procedure
8: function TRANSFORM(file)
9:     INITILIZE_EMPTY(list)
10:    for each pt_col_id ∈ file do
11:        af ← AFFROMCOL(pt_col_id)
12:        FINDREQ(list, af, pt_col_id)
13:        ENCFILELOADER(list, file)
14: end function
15: function FINDREQ(list, parent_af, pt_col_id)
16:    for each af ∈ maf | af.parent == parent_af do
17:        if af.req ⊈ af.avail then
18:            INSERTVERTEX(re_enc, af)
19:        else
20:            enc_field ← ENCSERVICE(pt_col_id, af.req)
21:            ADDTOLIST(list, enc_field)
22:            REPLACEAF(af, enc_field)
23:            FINDREQ(list, af)
24: end function
```

Multiple encryption fields. A potential overhead for the client is having to re-encrypt data in an encryption scheme appropriate for execution in the cloud. We use multiple encryptions for the same column whenever possible to minimize such

computations on the client side. For example in Listing 3.2 column $a0$ is used for comparison (line 3) and for addition (line 5). The naïve approach in this case would be to load column $a0$ in OPE to do the comparison and to re-encrypt it to AHE between lines 3 and 5 to enable addition. But this puts a computational burden on the client cluster, and requires sending data back-and-forth between the client and the cluster, thus increasing latency. Such a re-encryption can be avoided if we transform the source Pig Latin script such that both encryption forms for $a0$ are loaded upfront as two columns. We identify opportunities for such optimizations using the MAF abstraction. We describe next how the different encryption schemes to be loaded for a field a_i (a_i being a field in the source script and not an AF) can be identified. Once the req field is populated in MAF as part of analysis, we query the MAF for all AFs where a_i is part of the key. These AFs represent all usages of field a_i in the Pig Latin script. The set of values of req of these AFs and their child AFs represents the different encryption schemes required for field a_i. The transformation then generates the list of fields to be loaded from the encrypted version of the input file using metadata returned by the encryption service, and modifies the MET to use the newly loaded fields. This is described as function FINDREQ in Algorithm 4. Listing 3.3 shows the transformed Pig Latin script for our example. Note that field `a0` is loaded twice, under two encryption schemes.

Re-encryption and constants. AFs where the req encryption is not a subset of avail represent cases where a valid encryption scheme is not present to perform an operation. In such cases, we wrap the AFs in a specific `reencrypt` operation in the target script. This operation contacts the encryption service on the trusted client to convert the field to the required encryption scheme. Constants in MET are also transformed into encrypted form as required for the operation in which they are used. Note that Listing 3.3 shows the constant `10` encrypted using the OPE scheme to perform the comparison.

3.5 Implementation

In this section we provide details about our prototype implementation of SPR.

3.5.1 Overview

SPR is implemented as 6084 lines of Java code with two separate builds: (a) a client component which is deployed in the trusted space and (b) a cloud component which is deployed in the untrusted cloud (see Figure 3.1). The client component can generate and read both public and private keys used for encrypting and decrypting data as well as access the plain text input. The cloud component can only read the public keys used to encrypt data and has no access to private keys or plain text input. We use ØMQ [40] as the underlying messaging passing subsystem of SPR and the GNU multiple precision arithmetic library GMP [41] to perform fast arbitrary precision arithmetic operations. Both these libraries are implemented in native code and we use JNI to invoke the corresponding functions from Java. Our current version of SPR works with Apache Pig 0.11.1 running on top of Hadoop 1.2.1.

3.5.2 Program Transformation

Program transformation is the component responsible for generating the transformed Pig Latin data flow graph that works on encrypted data by performing the transformation process described in Section 3.4.4. In particular, if a transformed script contains re-encryption operations, the program transformation component assigns a unique *operation_id* to each such operation. This *operation_id* is included as part of the re-encryption function in the transformed Pig Latin script. Subsequently, the details needed to perform the re-encryption (for example the encryption schemes involved) are assigned to the *operation_id* which is then registered with the encryption service. When the script is executed, the encryption service uses the *operation_id* to retrieve the information needed to perform the re-encryption.

3.5.3 Encryption Service

The encryption service handles the initial data encryption as well as runtime re-encryptions. Initial data encryption are performed using Pig Latin scripts on the client-side where data is available as plain text. Encryption is done using UDFs that take each data item and convert it into a ciphertext corresponding to the required encryption scheme. Re-encryption requests are received by the trusted client over TCP sockets. These requests are added into blocking queues of worker threads. Worker threads process each request and return a response.

We implement randomized encryption (RAN) using Blowfish [42] to encrypt integer values, taking advantage of its smaller 64-bit block size, and use AES [43] which has a 128-bit block size to encrypt everything else. We use CBC mode in both of these cryptosystems with a random initialization vector. We construct deterministic encryption (DET) using Blowfish and AES pseudo-random permutation block ciphers for values of 64 bits and 128 bits respectively, and pad smaller values appropriately to match the expected block size. For values longer than 128 bits we follow the approach used in CryptDB [34] and use a variant of CMC mode [44] with a zero initialization vector. We perform order-preserving encryption (OPE) using the implementation from CryptDB. We use the Paillier [32] cryptosystem to implement additive homomorphic encryption (AHE), and the ElGamal [45] cryptosystem to implement multiplicative homomorphic encryption (MHE).

3.5.4 UDFs

The cloud component consists of custom UDFs and communication channels. Re-encryption operations are implemented as aggregation UDFs that take an entire relation as argument. This allows us to pipeline re-encryption requests as opposed to re-encrypting tuple by tuple. We also defined cryptographic equivalents of built-in Pig Latin UDFs like `SUM()`, or <.

3.6 Evaluation

This section evaluates our approach in terms of runtime performance and programmer effort. We also share lessons learned.

3.6.1 Synopsis

We evaluate four aspects of our approach:

(a) practicality to compute over encrypted data in terms of latency,

(b) programmer effort saved by our automatic transformations with respect to manual handling of (partially) homomorphic encryption,

(c) performance compared to existing solutions for querying on encrypted data, and

(d) scalability by running queries on large data-sets TBs.

3.6.2 Practicality: PigMix

We ran the Apache PigMix [36] benchmark to evaluate the practicality and performance of SPR . The evaluation was carried out using a cluster of 11 $c3.large$ nodes from Amazon EC2 [46]. The nodes had two 64 bit virtual CPUs and 3.75 GB of RAM. Input data with 3300000 rows (5GB) was generated by the PigMix data generator script and encrypted at the client side. We discuss these results here.

Figure 3.4 shows the results of the PigMix benchmark. On average we observe 3× overhead in terms of latency, which is extremely low compared to FHE (e.g., for simple multiplications we measured around 300000× slowdown with `HElib` [47]) which is currently still at least 1000′000×. We also observed that this overhead is correlated more towards size of the input data and not the actual computation. Some of the challenges we faced during the tranformation deals with limitations of Pig Latin in dealing with constants bigger than 32 bit `integers` or 64 bit `longs`. The range

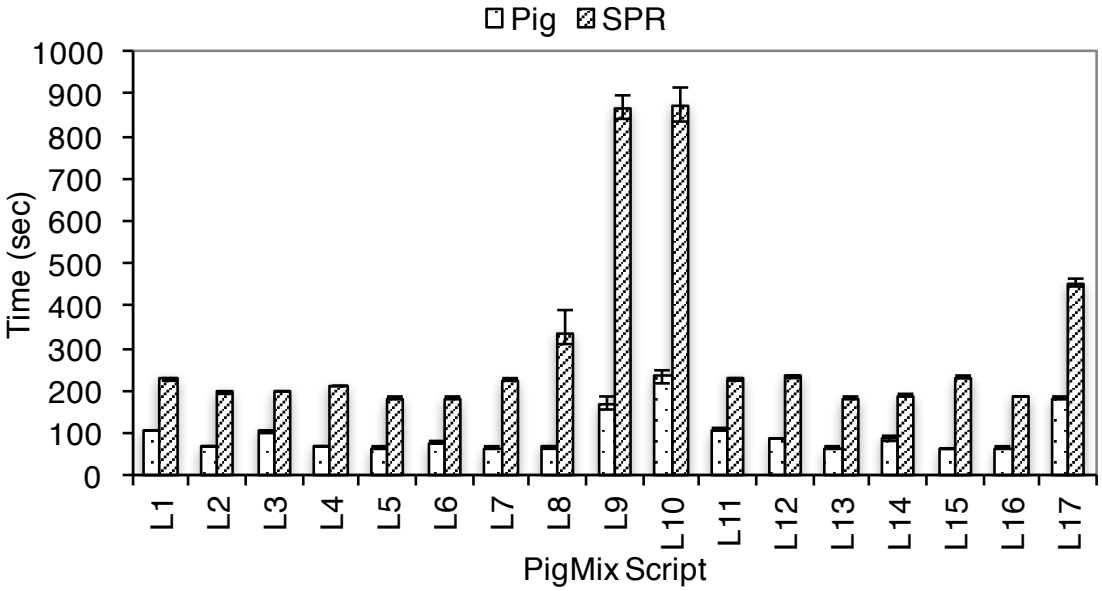


Fig. 3.4.: Latency of Pig Latin scripts in PigMix

of numbers that appear in the cipher text space may exceed what can be represented using `integers` and `longs` and we overcome this by representing numeric constants as strings and converting them to arbitary presicion objects like `BigIntegers` within UDFs. More details are outlined in our workshop report in [28].

3.6.3 Programmer Effort: PigMix

We now compare the number of tokens in the Pig Latin script written to be computed on plain text data with the Pig Latin script generated by our transformation. This gauges the additional work that a programmer would have to do in order to explicitly enable her script to compute on encrypted input data. We use the PigMix Pig Latin benchmarks for this comparison. Table 3.3 summarizes this result. As can be observed, our transformation generates Pig Latin scripts with 18% more keywords on average. For the scripts used in the PigMix benchmark, our transformation produces scripts which are between 5% and 36% more complex than the original scripts. As can be understood from our analysis, these additions are far from trivial.

Script	Original	Transformed	% Increase
L1	85	100	15
L2	75	86	11
L3	89	99	10
L4	62	70	8
L5	84	95	11
L6	66	74	8
L7	68	91	23
L8	61	72	11
L9	41	46	5
L10	46	57	11
L11	76	89	13
L12	134	158	24
L13	76	81	5
L14	73	79	6
L15	73	82	9
L16	62	80	18
L17	101	139	38

Table 3.3.: Number of tokens in original and transformed PigMix scripts

3.6.4 Performance Comparison: Weather Data

In order to compare our solution against the state of the art CryptDB system, we chose an application that analyses historical weather data set. The data set comprises of station names, dates and temperatures recorded on those dates on respective stations. The application issues common queries against an encrypted version of this data set. We compare the time taken by our solution to that of CryptDB. Cognizant of the fact that CryptDB was designed to address a different set of challenges than SPR we perform this comparison to understand how well our system and CryptDB performs when size of data increases. CryptDB is built on top of MySQL, a database that stores data in specialized data structures like B+ trees and optimized for data retrieval using indices and supports transactions including updates. In contrast, SPR is built on top of the Pig runtime, which specializes on reading from flat files. Furthermore, the Pig runtime depends on MapReduce, which has an intermediate stage which involves writing data to disk before reducers read it remotely over TCP chan-

nels. Conversely, by using MapReduce, Pig can leverage the processing power and memory of multiple machines while CryptDB is confined to a single node. We perform three comparisons: (1) how our solution performs on a fixed size cluster as size of input data set varies, (2) how our solution scales by increasing the number of nodes in the cluster and (3) what is the monetary cost of performing the computation (in Amazon EC2). The results presented here compare the latency of running a query that finds the average temperature for each station using encrypted data. We compare the time taken by CryptDB (single node) with the latency of running equivalent Pig Latin script on a Hadoop cluster with 4 nodes (3 worker nodes and 1 control node). We used *m3.medium* nodes on Amazaon EC2 for all our evaluations. As can be observed from Figure 3.7 SPRis able to exploit the parallelism offered by multiple nodes and scales well. CryptDB was designed for high throughput in terms of queries per second and not for processing high volumes of data. This can be observed by the increase in time taken for the queries over CryptDB as the number of records increases. For 15 million rows, SPR is 2.7× faster than CryptDB and shows much slower increase in job completion latency as the number of rows increases. Figure 3.5 shows how time taken by SPR changes as the number of nodes in the cluster increases. We show the time taken by CryptDB running on one node as a base line. We can observe that running on encrypted data shows trends comparable to regular Pig Latin jobs and latency reduces as the number of nodes increases. It may seem surprising that the time taken by the Pig Latin script running on one node was quite comparable to the time taken by CryptDB. The reason for this is that for both CryptDB and SPR the time taken is bound by the number of cryptographic operations that the CPU has to perform; which is the same for both systems. Furthermore, in order to find the average temperature, we find the `AHESUM` and `COUNT` of temperature readings for each group. Because both the UDFs implements `combiner` and `algebraic` interfaces, and the number of distinct groups we have is not high, most of the output generated by the map phase is combined before being pulled by the reducer. Also, even with a single node, we have two mappers and one reducer running simultaneously providing some

level of parallelism. These factors limit the overhead caused by data transfers and the latency is predominantly determined by the time taken by the CPU to perform a fixed number of operations.

We also look at the monetary cost of running our computation and compare it with the cost of running CryptDB. We compute the cost by summing up the total amount charged by Amazon for all the nodes used for running our experiments. We present the calculated cost in Figure 3.8. As can be observed, we incur a (surprisingly) similar overall cost despite completing the job up to 2.7× faster.

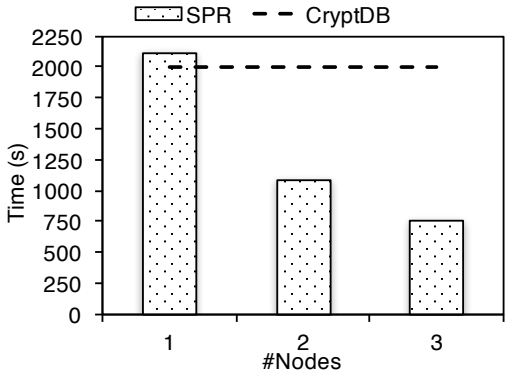


Fig. 3.5.: Comparison of SPR and CryptDB runtime latencies with varying #nodes

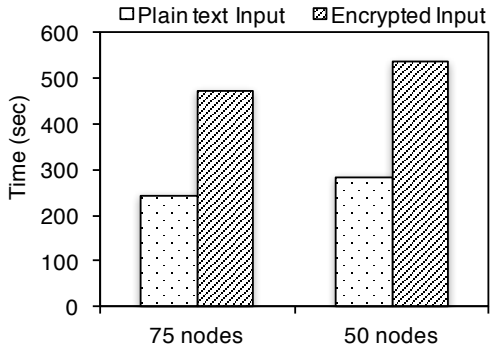


Fig. 3.6.: Latency to run word count on encrypted and plain text input

3.6.5 Scalability: Word Count

In order to test the efficacy of our solution on big data, we ran experiments using 1 TB of data obtained from Wikipedia dumps [48]. Since the data was in XML format, we first ran a Pig Latin script to obtain the text section from the file. Next, we tokenized and encrypted all the words, then executed our word count script on the encrypted words. Lastly, we decrypted and stored the top 5 words. In addition to an encrypted word count, we also ran our word count script on the plain text. The time for SPR does not include the time needed to encrypt the data, since it is assumed

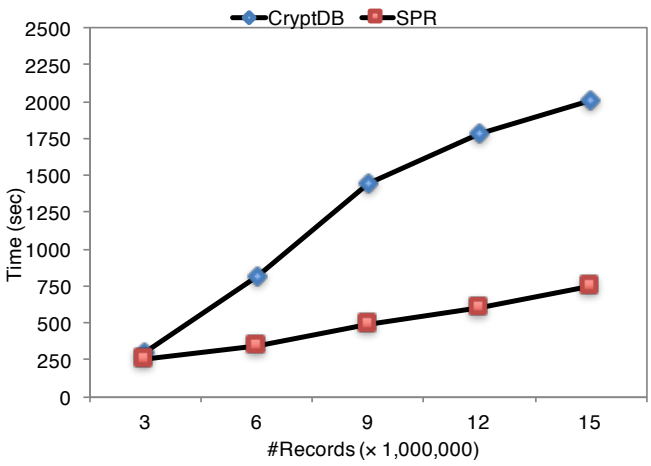


Fig. 3.7.: Comparison of SPR and CryptDB runtime latencies with varying #records

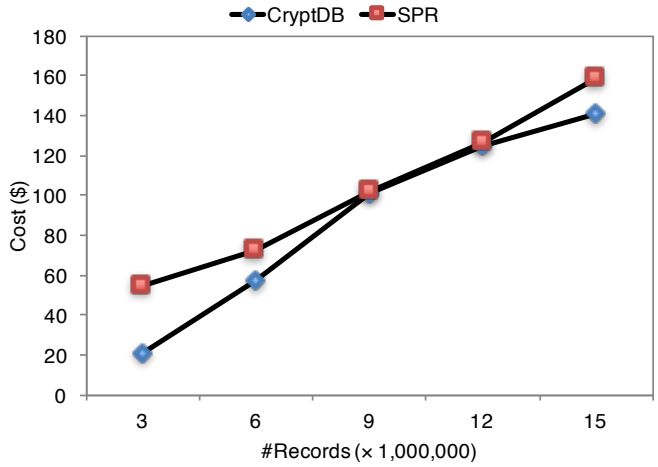


Fig. 3.8.: Comparison of SPR and CryptDB execution cost

that the user will have the encrypted data on the untrusted cloud. We measure the latency of running word count on the encrypted input. We use Hadoop clusters of 50 and 75 nodes and summarize the results in Figure 3.6. We observe that for both jobs, reduction in latency as number of nodes increase follows the same trend.

3.6.6 Threats to Validity

In evaluating our system, we have considered several threats to validity. Multi-tenancy and the virtualized nature of Amazon EC2 can lead to variations in latency measurements. We thus ran all our evaluations several times (5 typically) and took the average to counter variations. The variations in latency we observed in different runs of the same Pig Latin scripts were relatively low (4%). For CryptDB we observed a variance of up to (10%). We have tried to use benchmarks that are representative, meaningful and used by other publications. PigMix, for example is a standard benchmark used by Apache on every new release of the Pig runtime system. While the data sets we used for it are only in the GB order, such sizes are not atypical for big data in fact [49], and used also by others [35]. One factor to consider with respect to big data analysis is that when volume of data becomes very high, characteristics on input data may heavily determine the time taken for completion.

3.7 Discussion

Two big drawbacks of computing on encrypted data is bloating of data size and additional CPU cycles required to do more complicated arithmetic. For example, if we are using Paillier with 1024 bit length keys, a 64 bit `long` may get converted to a cipher text 2048 bits in length (an increase of 32 times) and an addition in plain text is replaced by multiplication of cipher text followed by a modulo operation in Paillier. Unfortunately we cannot avoid paying a higher price in terms of storage and computation based on state of art cryptosystems. Another aspect of security that we do not consider is that even though data is encrypted, an attacker can still see data access patterns. This includes frequency of data access, groups of data accessed together, operations done on data etc.

4. CONFIDENTIALITY IN STREAM PROCESSING SYSTEMS

This chapter presents our solution for performing confidential data analytics over streaming data (published in [50]).

4.1 Overview

The ubiquity of computing devices is driving a massive increase in the amount of data generated by humans and machines. With the advent of the Internet of Things (IoT), many more billions of devices are expected to continuously collect sensitive data (e.g., location data, personal health data) and compute on it. Due to limited storage and computation capacity available on IoT devices, the current *de facto* model for building IoT applications is to send the data gathered from physical devices to the cloud for both computation and storage (e.g., SmartThings [51], Nest [52]). Many IoT applications therefore leverage the cloud to compute on data streams from a large number of devices. For example, to compute variable tolls or to identify highway accidents, a smart city application may collect vehicle license plate numbers, speed, and location information at the cloud.

Due to the sheer amount of the streaming data, building a private cloud infrastructure is very expensive compared to using a low cost public (untrusted) cloud infrastructure such as Amazon EC2 or Microsoft Azure. Therefore, public clouds are typically used for processing continuous queries including on sensitive data. However, this trend is leading to increasing concerns over data confidentiality, and is becoming one of the major factors preventing more widespread adoption of IoT solutions. For instance, a recent study, among 2,062 American consumers, shows that the top concern is *"Who is seeing my data?"* [53].

One way to mitigate these concerns is to encrypt data at the source (i.e., IoT device), and to solely use cloud infrastructure for storage purposes (e.g., Bolt [54]). Thus, as long as encryption keys are maintained securely by consumers, their data remains secured. While this approach addresses the aforementioned confidentiality concerns, *all computations* need to be performed in trusted environments. This solution strongly limits the computational capabilities available for IoT solutions.

A promising approach to tackle these issues is to use *homomorphic encryption* and execute all operations over encrypted data. However, *fully* homomorphic encryption (FHE) is prohibitive in practice [55], causing slowdowns by an order of $10^9\times$. An alternate, practical approach is to use less expensive *partially* homomorphic encryption (PHE) to execute specific operations over encrypted data. Yet, existing solutions use a storage system to this end. For instance, the seminal CryptDB [34] was implemented on top of MySQL, while MONOMI [56] and Talos [57] were implemented on top of Postgres. These database-centric solutions are not a good fit for many IoT applications because IoT applications are typically implemented as continuous queries in a stream processing system.

A straightforward application of PHE to existing stream processing solutions to support computations over encrypted data is however unlikely to be practical: (G1) Dozens of PHE *schemes* exist, varying by operations supported, efficiency, size of encrypted data etc.; IoT application developers do not necessarily possess sufficient in-depth knowledge of crypto(graphic) schemes to judiciously select among these. For resource-constrained IoT devices, efficiency incurred by individual crypto systems is a major concern. (G2) Moreover, encryption typically increases the size of input data, with different factors for different crypto systems. Specific optimizations are required to reduce this size difference to make the solution practical. (G3) Furthermore, due to the limitations of PHE schemes, special handling is required for variable initializations and constants in a program. (G4) With applications running continuously and potentially infinitely, secret keys used for encryption need to be updated periodically or on demand (e.g., when there is a compromise). Such updates on the IoT devices

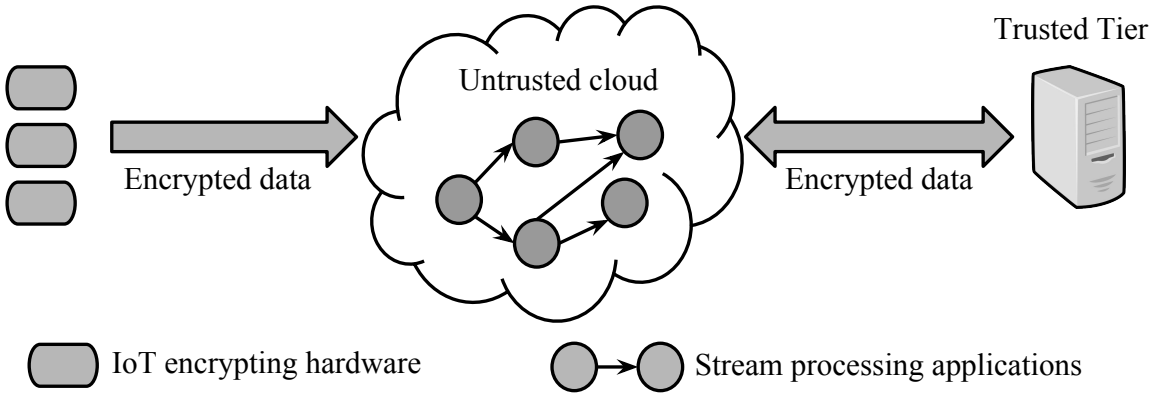


Fig. 4.1.: STYX overview

should be made transparently to the IoT application and should not lead streaming queries to miss results. (G5) Processing continuous queries typically involves a pipeline of computing tasks and each task may have one or more instances running concurrently. The *deployment profile* which maps task instances to VMs in the cloud should make balanced use of resources to avoid bottlenecks. While some optimization heuristics are known, they do not consider encryption which shifts bottlenecks. (G6) Finally, as hinted to by their names, PHE schemes do not support arbitrary operations. Unsupported operations have to be performed on the trusted client side in a plain text form. Consequently, either the query processing can continue on the trusted client side or the intermediate results can be re-encrypted to the schemes required by subsequent operations and continued in the cloud. Deployment profiles must be cognizant of such re-encryptions.

This paper presents STYX, a novel programming abstraction and managed runtime system, that leverages PHE to provide confidentiality for IoT applications delegating online streaming jobs to the public cloud. STYX operates on streaming data without revealing any plain text information to the untrusted cloud. Figure 4.1 gives a high level overview of STYX. A user designs, implements, and initiates the stream analysis program that runs in the untrusted cloud. IoT sensors automatically encrypt generated data before emitting them in the stream for analysis. Additional streams of

encrypted private data required for analysis can be sent independently from a trusted tier maintained by the user.

To perform analytics in the untrusted cloud over encrypted data (whilst addressing G1-G6), STYX: 1. enables programmers to develop applications using the STYX API for typical plain text streams and automatically transforms the application to work with encrypted streams. This means that developers will only focus on application logic and not on the details of the underlying crypto systems (G1); 2. handles variable initializations and constants correctly through transformations (G2 and G3); 3. utilizes PHE-specific optimization techniques (e.g., field masking) to reduce encrypted data size and provides efficient implementation of these techniques so they can run on IoT devices (G3); 4. supports transparent update of secret keys on IoT devices (G4); 5. deduces the best way to deploy an application through an analytical modeling module (G5); and 6. is capable of executing the remainder of the computation in the trusted tier or re-encrypting a data stream (or parts of it) to enable further computation in the public cloud if a given sequence of computations cannot be performed due to PHE limitations (G6).

This paper makes the following contributions:

- We introduce a secure stream abstraction that exposes a high level API through which programmers can express programs that can be executed in the public cloud in a way preserving confidentiality without having to know the details of underlying crypto systems. We also introduce STYX, a system that support this API and addresses related challenges.

- Describe how STYX analyzes programs written using the STYX API and identifies the computations that can be executed purely on encrypted data and the computations that cannot, due to the limitations of PHE. STYX maximizes the amount of computation performed in the cloud by splitting computation between the untrusted cloud and a small number of trusted nodes while automatically performing required

Table 4.1.: STYX crypto systems

Crypto system	Operation
Random AES	–
Deterministic AES	$x_1 = x_2 \iff E(x_1) = E(x_2)$
Boldyreva et al. [58]	$x_1 > x_2 \iff E(x_1) > E(x_2)$
Paillier [32]	$x_1 + x_2 = D(E(x_1)\psi E(x_2))$
ElGamal [45]	$x_1 \times x_2 = D(E(x_1)\psi E(x_2))$

re-encryptions. Fast serialization techniques and encryption pre-computation are two techniques used to assure the efficiency of STYX.

- Propose a heuristic that analyzes resource availabilities and requirements and generates a deployment profile that optimizes cloud usage.

- Evaluate the implementation of STYX on multiple benchmarks and case studies. Our results indicate that STYX can be used to express many real-world IoT applications, including variable smart city toll applications, while ensuring confidentiality transparently and keeping low overhead.

The remainder of this paper is organized as follows. Section 4.2 presents background information on PHE and continuous queries. We give an overview of our solution in Section 4.3. Sections 4.4 and 4.5 present the design of STYX and its runtime system respectively. We discuss implementation details of STYX in Section 4.6 and present our evaluation results in Section 4.7.

4.2 Background

In this section we present background information on partially homomorphic encryption (PHE) and systems that support continuous queries.

4.2.1 PHE

A crypto system is said to be *homomorphic* (with respect to certain operations) if it allows computations (consisting in such operations) on encrypted data. If $E(x)$ and $D(x)$ denote the encryption and decryption functions for input data x respectively, then a crypto system is said to be homomorphic with respect to ϕ if $\exists\psi$ s.t.

$$D(E(x_1)\psi E(x_2)) = x_1\phi x_2$$

E.g., a crypto system is said to provide *additive homomorphic encryption* (AHE) when $\phi \leftarrow +$. Other homomorphisms are with respect to operators such as "$\times$" (*multiplicative homomorphic encryption* – MHE), "$\leq$" and "$\geq$" (*order-preserving encryption* – OPE) or equality comparison "=" (*deterministic* encryption – DET).

4.2.2 Continuous Queries

The core abstractions offered by systems that support continuous queries are *streams*, *tuples*, and *fields*. Briefly, a single logical data unit is a field, one or more fields make up a tuple and an unbounded stream of tuples is denoted as a stream. The tuples in the stream are processed in a distributed fashion. Application logic is arranged as a directed graph where vertexes of the graph are computation components and edges are streams that represent the data flow between components. Application programmers write application logic for the vertexes of the graph. A subset of these vertexes are also designated as *source vertexes*. These source vertexes act as entry points for data into the graph. Source vertexes typically read data from a queue, log file, or external subscriptions. As data is generated in real time and added to the queue, it is picked up by the source vertexes and forwarded down the graph for processing according to a specified grouping clause provided by the programmer.

Figure 4.2 shows a graph with four vertexes with $v1$ designated as the source vertex. The figure also shows streams $s1, s2, s3$ and $s4$. Each vertex of the graph

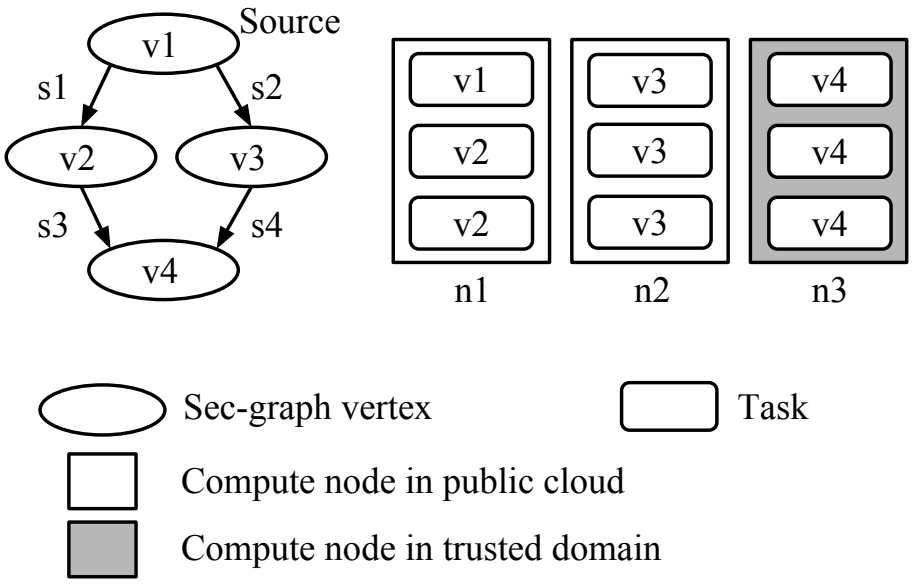


Fig. 4.2.: STYX graph and tasks

may have multiple runtime instantiations called *tasks*. In Figure 4.2, vertex $v2$ has two tasks running in Node 1 (a *node* represents a virtual or physical machine) and $v3$ has three tasks running in Node 2. We refer to this assignment (i.e., a specific number of tasks to each vertex) the *deployment profile* of the graph.

The stream emitted by each vertex is declared explicitly in the vertex itself. Once all vertexes of the graph are designed, the graph is assembled by defining the input stream of each vertex and specifying grouping clauses.

4.3 STYX Overview

In this section, we give an overview of STYX's threat model, program abstractions, and runtime execution flow.

4.3.1 Threat Model

STYX provides strong confidentiality guarantees against an adversary with full access to servers in the cloud: the adversary can have root access to cloud servers.

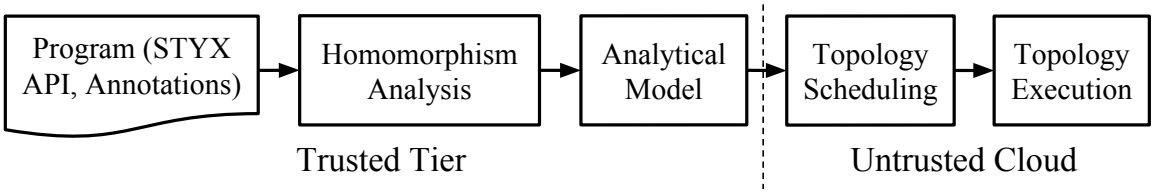


Fig. 4.3.: STYX execution flow

This means that the adversary may inspect all data in the server and even examine RAM or CPU caches of physical machines. STYX's goal is to preserve confidentiality against such an adversary. We note that STYX's goal is to preserve confidentiality and not integrity or availability. We assume that STYX has access to a set of limited but trusted resources outside the cloud. As we shall see shortly, this environment is leveraged to perform certain computations.

4.3.2 STYX Abstractions

One of the main challenges of computing over encrypted data is that the application developer needs to have a detailed understanding of each crypto system used to encrypt fields of a stream. Adoption of PHE and even FHE for generic application development will depend on the ease with which a programmer can incorporate the properties offered by the crypto system into their regular programming tasks. STYX tackles this problem by offering simple programming abstractions to express and operate on encrypted data streams. E.g., fields representing sensitive data are defined using the *secure field* abstraction, irrespective of the kind of operation that needs to be performed on them or the underlying crypto system used to represent the field. STYX relieves the programmer from the task of identifying the crypto system required for the operation that the programmer needs to perform. To this end, a programmer simply annotates the stream with the desired operation and STYX deduces the crypto system that needs to be used at the source IoT devices.

4.3.3 Execution Flow

We now give an overview of the STYX execution flow. Figure 4.3 outlines the steps followed by STYX from the time a STYX graph is designed. Application programmers use the STYX API to design a graph which contains the application logic. STYX then performs *homomorphism analysis* on the graph to identify the crypto systems required to execute the graph in a confidential manner. The details of this analysis are communicated to the IoT devices which generate key pairs corresponding to the crypto systems. A detailed explanation of these steps are given in Section 4.4.

Next, STYX analytically identifies the number of tasks required for different vertexes. It then schedules the graph for execution. STYX leverages the idea that often users have some limited (but trusted) computing resources. We refer to these resources as the *trusted tier*. The compute resources in the cloud, though potentially unlimited for practical purposes, are untrusted. STYX utilizes the trusted tier for application development and compilation and uses the cloud for the deployment phase. In deployments that require resources from the trusted tier, STYX tries to minimize their usage. The detail of deployment steps are presented in Section 4.5.

4.4 STYX Secure Streams

In this section, we describe the programming abstractions used in STYX and explain how these abstractions are leveraged for improving performance.

4.4.1 Program Model

Application programmers use the abstractions provided by our STYX Java API to specify a graph. In what follows, we explain the abstractions that are new to STYX over typical stream processing systems such as Storm. These abstractions are also summarized in Table 4.2.

```
1  /** Code executed when a vertex receives a tuple */
2  @EncOperations(operations = { "{eq}", "{sum}" })
3  public class StyxSumVertex extends StyxVertex {
4      SlotBasedSum<SecField> slidingWindowGroupSums =
5              new SlotBasedSum<SecField>(60);
6      public void execute(Tuple tuple) {
7          //if timing tuple ...
8          emitCurrentWindowCounts(slidingWindowGroupSums);
9          //else ...
10         SecField group = FieldOper.getField(tuple, 0);
11         SecField value = FieldOper.getField(tuple, 1);
12         slidingWindowGroupSums
13                 .updateSum(group, getCurTimeSec(), value);
14     }
15 }
16 /** Track sum of values per group in each time slot */
17  public class SlotBasedSum<T>  {
18     ...
19     public void updateSum(T group, int slot, SecField value) {
20         SecField[] sums = objGroupSum.get(group);
21         if (sums == null) {
22             sums = new SecField[this.numSlots];
23             init(sums, value);
24             objGroupSum.put(obj, sums);
25         }
26         sums[slot] = SecureOper.add(sums[slot], value);
27     }
28 }
```

Listing 4.1: STYX code for finding the sum of each group in a sliding window

STYX API. STYX exposes its programming abstractions using an API described below.

`SecField`. Programmers use this class to realize the secure field abstraction that refers to a confidential input field. E.g., programmers can get a reference to the first value in a tuple as shown in Listing 4.1 Line 10. Secure fields can also be initialized by reading an encrypted value directly from an input stream.

`SecOper`. Secure operations are operations provided by STYX that allow programmers to implement standard operations like `multiply`, `add`, `compare` and `equal`. Programmers use the `SecOper` class to perform these operations. Secure operators take secure fields as input and return a secure field or Boolean value as result. E.g., in order to perform an addition, programmers may write code similar to

`SecField result = SecOper.add(operand1, operand2)` in order to add two secure fields, `operand1` and `operand2`.

`StyxVertex.` This base class is extended by programmers to express the computation in a vertex of the graph. This class provides the `execute()` method which is invoked by STYX when a tuple arrives at a vertex for execution.

Table 4.2.: STYX abstractions

STYX API	Function
`SecField`	Confidential field
`SecOper.add(f1, f2)`[1]	`f1`$\times$ `f2` *mod* `n`^{2}
`SecOper.multiply(f1, f2)`[2]	`f1.a` $\times$`f2.a`
	`f1.b` $\times$ `f2.b`
`SecOper.compare(f1, f2)` [3]	`f1` $\leq$ `f2`

[1] `n` represents the public key used to encrypt `f1` and `f2`.
[2] MHE scheme cipher text contains two components denoted here by `a` and `b`.
[3] OPE scheme requires comparisons over integers 128 bits and higher.

Stream annotations. Along with using the STYX API, programmers also annotate each stream with the operations they want to perform on that stream. E.g., `StyxSumVertex` in Listing 4.1 shows a secure stream with two fields, and the vertex implementing the standard `Group By..Sum` operation. For this, the programmer uses the definition `operations = { "{eq}", "{sum}" }`. This shows that the first field is used in equality comparisons and the second field in summing. These annotations enable our compile-time graph analysis to identify the crypto systems for performing these operations and to apply additional performance improvement techniques introduced in Section 4.4.2.

Example. Listing 4.1 shows code snippets used in a STYX vertex class. The code is part of a graph that keeps track of the sum of values in different groups within a sliding window (last one minute in this example). The input tuple contains two fields: the group name and the value for that group (Lines 10 and 11). The code

shown retrieves the group and value fields from the input tuple (Lines 10, 11) and updates the sum for that group's current time slot (Line 12) with the value. Every time the vertex receives a timing tuple, signifying a minute has elapsed, it emits the sum of all groups in the current sliding window (Line 8). The object maintaining the sliding window internally contains a map and updates the group's sum every time the `updateSum()` method is called using STYX's `SecureOper.add()` method (Line 26). Note that Line 2 also shows the stream annotations.

Listing 4.2 shows just the function `updateSum()` from Listing 4.1 written without using STYX abstractions. As can be seen, this requires the programmer to explicitly read the public key (see Line 2), and perform the exact AHE evaluation operation for addition – multiplication followed by modulus using the square of the public key if using the Paillier crypto system as in Line 10.

```
1  public class SlotBasedSum<T>  {
2     BigInteger publicKey = readPubKey();
3     public void updateSum(T group, int slot, BigInteger value) {
4         BigInteger[] sums = objGroupSum.get(group);
5         if (sums == null) {
6             sums = new BigInteger[this.numSlots];
7             init(sums, "AHE");
8             objGroupSum.put(group, sums);
9         }
10        sums[slot] = sums[slot].multiply(value)
11                    .mod(publicKey.multiply(publicKey));
12    }
13 }
```

Listing 4.2: Code for finding the sum of each group in a sliding window without STYX abstractions

4.4.2 Processing Secure Streams

We now give details on how we tackle the challenges introduced in Section 1 when processing continuous queries over encrypted streams.

Field masking. Computing on encrypted data introduces the additional challenge of dealing with operands with increased sizes. For instance, an addition of two `long`

operands (each 64 bits) in plain text may transform into an operation over 1024 bit operands in the encrypted data stream. This means a factor of 16 increase in the operand size. Typically, in stream processing systems, the source vertex receives all the fields in the stream, irrespective of a field being used or not. For plain text program graphs, this is usually not a substantial overhead and the additional computation required for removing unused fields may not always offset the improvement that is observed. When the computation happens over encrypted data, filtering out fields will have a much more significant impact because of the size of the fields. E.g., consider a stream with two fields similar to the stream used for `Group By..Sum` in Listing 4.1. If there is a continuous query which finds unique groups, the second field will be unused. Simply removing the unused field reduces the size of a tuple from 160 bytes to 32 bytes. STYX performs this unused field removal automatically using field masking. Considering the fact that an unused field may be at any index within a tuple, if we simply drop the field, program logic that accesses the field after it has been dropped may fail. To avoid this problem, we mask unused fields by not removing them, but replacing them with `nulls`. To identify unused fields, STYX relies on the stream annotations described in Section 4.4.1. For each vertex, we identify fields for which no operations are specified. Our masking process itself is very lightweight. Since we have information about fields to be masked at compile time, we update the STYX runtime with this information. STYX then suppresses the emission of the masked fields.

Key management. If a device is compromised, it will need to update its private key along with relevant security patches. There also may be a need to periodically update or revoke keys in an IoT device. If a continuous query aggregates data over a sliding window, it is not possible to perform these operations without disrupting the output. This is because if we decide to switch over to a new key at time t, aggregations in the sliding window that span both sides of time t will contain tuples encrypted with both old and new keys and will fail. STYX supports such key changes without disrupting the output. When a key change is initiated, tuples are emitted under the old and

new key for the time span of the sliding window in the application graph. When the `StyxVertex` base class detects a key change in the stream, it creates a new instance of the application vertex class to process the stream. The stream encrypted with the new key is channeled into the new instance of the vertex class, while the stream encrypted with the old key gets processed as before. STYX suppresses emissions from the new vertex instance until the new instance contains tuples spanning the full length of the sliding window. At this point, the old instance of the application vertex class is discarded and the stream from the new instance is emitted.

Initialization and constants. Now we address the challenge of variable initializations. Standard operations like `sum = sum + value` usually require `sum` to be initialized to 0 or another constant value (say `k`) before being executed. When the same operation is performed over encrypted data, `sum` must be initialized to the encrypted value of the `k`, and further, encrypted using the same crypto system and key as `value`. STYX handles this by allowing the runtime to request constants from the trusted tier. E.g., STYX code like `SecField sum = SecureOper.getConstant(k, value)` is used to initialize a variable to `k` (used within the `init(sums, value)` method in Listing 4.1, Line 23). STYX uses its internal meta data to identify that the field `value` is derived from the second field in the input stream and requests the trusted tier to return `k` encrypted using the same crypto system and key as variable `value`. Note that the number of potential encryptions of a constant is bound by the number of unique fields across all streams in a graph. This allows the trusted tier to pre-compute and cache commonly used values and reduce latency.

Identifying encryption schemes. This step identifies the crypto systems that are required for the various fields based on the operations that the application wishes to perform on those fields. To apply these inferences, STYX first has to identify different streams and their grouping clauses in the application logic. This can be derived from the graph declaration as explained in Section 4.4.1. Secondly, we need to identify the operations performed on each stream. STYX derives this from program annotations in each vertex class in the graph. Once STYX derives the distinct streams and

operations to be performed on those streams, we can proceed similarly as in our prior work [29] to infer the crypto systems required to execute the graph.

In what follows, we briefly summarize how these techniques work. We start by constructing an expression tree where fields in tuples form the leaf nodes. Operations performed on those fields form the non-leaf nodes. For STYX graphs, we use field annotations (as specified in Section 4.4.1) to determine the non-leaf, operator nodes. For each operator, a lookup table identifies the crypto system of the operands and result of the operator. Our goal now is to identify the crypto system in which all the leaf nodes (fields) should be encrypted. This can be done by identifying the parent operator node for each leaf node and using the lookup table to identify the type of crypto system required for operands for that operator node. There can also be a mismatch between parent and child operator nodes if they do not belong to the same crypto system. In this case (unlike in [29]) a re-encryption operator is inserted in between, which converts the stream from one crypto system to the other. These re-encryption nodes are marked to be executed on the trusted tier so that the scheduler can place the tasks correctly.

Automatic re-encryption. Once the analyzer determines the crypto systems required for each stream, it may turn out that some operations cannot be performed over the available homomorphic crypto systems in the cloud. To get around this issue, STYX may decide to either perform those operations in the trusted tier or re-encrypt the stream in the trusted tier. For re-encryption, STYX inserts special *re-encryption vertexes* into the graph and marks them so they get scheduled on the trusted tier only.

Encrypting public streams. Though the IoT devices handle most of the encryptions, sometimes the programmer may have plain text data from public streams like stock quotes which are part of an application logic. Though these data are public, we may need to encrypt them because if they are to be combined or compared with private data already encrypted for computing in the cloud, the public data should also be encrypted under the same key for such operations to succeed. For easily

Given

S: $[s_1, s_2 \dots s_n]$ Available slots

V: $[v_1, v_2 \dots v_m]$ Vertexes

A: $\{[a_{11} \dots a_{1m}], \dots, [a_{m1} \dots a_{mm}]\}$ where,

a_{ij}: Output from v_i that goes to v_j for unit input

C: $[c_1, c_2 \dots c_m]$ where

c_i: Relative load on any s_j processing v_i

Unknowns

d_i: Input consumed by single instance of i

D_i: Input consumed by all instances of i

T: $\{t_1, t_2 \dots t_m\}$ where

t_i: Number of slots for v_i

From definitions

$D_1 \leftarrow d_1 \times t_1$

$\forall 1 < i \leq m,$

$D_i \leftarrow d_1 \times t_1 \times \{A_{1i} + A_{1i}^2 + \dots A_{1i}^l\}$ where,

l is the length of longest path between v_1 and v_i

Linear program constraints

$\forall 0 < i < m, t_i \geq 1$ All v_is are allocated

$\forall 0 < i < m, D_i < c_i \times t_i$

Linear program objective function

$\max_t(D_1)$

Fig. 4.4.: STYX heuristics

integrating public data into a program, STYX provides special vertex classes that encrypt plain text streams.

4.5 STYX Deployment

In this section we describe how graphs are deployed into the STYX runtime.

4.5.1 Deployment profile generation

As defined in Section 4.2, the number of runtime tasks assigned for each vertex in the graph is called the deployment profile of the graph. A good deployment profile is required to avoid bottlenecks and ensure good resource utilization.

Utilization. To reason about the effectiveness of deployment profiles, we first define *utilization*. Utilization of a vertex for a time interval is defined as the amount of time the vertex spends processing in that time interval. E.g., if a vertex spends 5 minutes in a 10 minute interval processing tuples and the rest of the time waiting for tuples to arrive, then it has utilization of 0.5. As utilization of a vertex approaches 1, we can assume it is starting to become a bottleneck. Good resource utilization is usually achieved by the programmer explicitly specifying the number of tasks for each computation vertex. Programmers are perfectly suited to do this as they understand, via application logic, which vertexes handle more data or computation, and can correspondingly allocate more tasks for those vertexes. In STYX, when the computation graph is transformed and operations are converted to their cryptographic equivalents, the utilization of a vertex changes substantially. This means that programmers need to thoroughly understand overheads of each crypto system, which goes against STYX's design goals.

Heuristic. We propose a linear programming based heuristic which automatically converts the deployment profile for a plain text graph into an optimized deployment profile for the corresponding STYX graph. Figure 4.4 shows the formal representation of the heuristics that we use. S represents the slots available for instances to use and V represents the vertexes in the graph that needs to be allocated. A slot is typically a Java virtual machine (JVM) or an executor thread within a JVM. We assume all slots have the same processing capacity. The matrix A represents how much each vertex amplifies its input. A is derived by executing the plain text graph on sample data. To compute the amount of data arriving at a vertex, we consider all paths of varying lengths that end up at that vertex from the source. The A matrix gives the amount

of data at vertexes which are one edge away from the source (for unit input). To find data arriving at the vertex i (represented by d_i) through paths of length 2 and higher, we compute the power matrix of A represented in Figure 4.4 as A^2, A^3 etc. The vector C represents the load on each vertex relative to one another. C is derived by first inverting the number of instances for each vertex (from the deployment profile) in the plain text version of the graph and then scaling it with respect to the crypto operations performed by the vertex.

For example, assume that a programmer specifies the number of instances for each vertex as $v1:1, v2:3, v3:2, v4:6$ for the plain text graph presented in Figure 4.2. Thus, we start with the vector $\{1, 1/3, 1/2, 1/6\}$ or simply $\{6, 2, 3, 1\}$. The intuition here is that for vertexes that come under heavy load, the programmer will allocate a higher number of instances in the deployment profile to accommodate the load. At the next step, we scale down the value of each element in the above vector based on a reduction factor. This reduction factor is derived empirically based on our observations. We use a reduction factor of 3 for AHE schemes, 2 for DET and 6 for re-encryption nodes. Consequently, and based on Figure 4.2, if $v3$ receives a stream ($s2$) with a field encrypted under AHE, we scale down the value corresponding to v_3 to 1, after which we arrive at $C \leftarrow \{6, 2, 1, 1\}$. After scaling down each element we get the C vector which can be used in the linear program.

In Figure 4.4, T represents the deployment profile, and t_i represents the number of slots allocated to execute vertex v_i. Our target now is to derive each t_i. To this end, we define two sets of constraints. The first set of constraints ensures each vertex is allocated to at least one slot. The second set of constraints ensures that for all vertexes, the load (described by c) is less than the capacity of the nodes to process it. Under these constraints, we maximize the amount of data that can be consumed at the source vertex.

4.5.2 STYX scheduler

The primary responsibility of the STYX scheduler is to decide on which host machines vertexes of the graph will be executed. The STYX scheduler is provided with two lists of hostnames, one that lists hosts in the untrusted cloud, and another that lists hosts in the trusted client environment. The scheduler reads the graph annotation to identify where each vertex must be executed.

For components that need execution in a trusted environment, the scheduler sends the appropriate class files to the worker instances running on the trusted side. The trusted side workers have access to private keys that are required for encryptions or crypto system transformations. The workers in the untrusted cloud only see the encrypted data and have access only to the public keys required to perform the homomorphic operations.

We note that the scheduler service can also run in the untrusted cloud. An attacker can try to manipulate the scheduler in the following two ways: (i) by trying to execute trusted vertexes in the untrusted cloud and (ii) by trying to execute untrusted code in the trusted tier. The former way does not compromise confidentiality since the untrusted cloud does not possess the private keys required to reveal the plain text data. However, the latter can compromise confidentiality if the attacker is successful in executing malicious code that retrieves private keys or read data when they are in plain text while being re-encrypted. To avoid this, a hash of the vertexes to be executed in the trusted tier is generated before deployment. When tasks are delivered to the trusted tier for execution, the trusted tier first computes a hash of the task class, and compares it with the hash generated before deployment. Execution proceeds only if there is a match.

4.6 Implementation

In this section we lay out some of the implementation details of STYX. STYX's processes in the cloud are implemented by modifying Apache Storm [59]. Storm is

an online, distributed computation system. Application logic in Storm is packaged into directed graphs called *topologies*. Vertexes of the topologies are computation components and edges represent data flows between components. There are two types of components in Storm: (i) *spouts* that act as event generators, and (ii) *bolts* that capture the program logic. In other words, spouts produce the data streams upon which the bolts operate. Modifications to Storm are limited to implementing a new scheduler (by overriding the `IScheduler` interface) and changes to the way a Storm topology is submitted (`StormSubmitter` and related classes). These changes add an additional 1031 lines of code to Storm. The programming interfaces and runtime cryptographic classes that allow computations over encrypted data are packaged as a separate `jar` library, implemented in 3633 lines of Java code. The cryptographic classes make use the GNU multiple precision arithmetic library GMP [41] to perform fast arbitrary precision arithmetic operations invoked using the Java native interface (JNI). Randomized encryption (RAN) is implemented using AES [43] with CBC mode with a random initialization vector. Deterministic encryption (DET) is implemented using an AES pseudo-random permutation block cipher following the approach used in CryptDB [34] that uses a variant of CMC mode [44] with a zero initialization vector. The Boldyreva et al. [58] crypto system is used as our OPE scheme implementation. The Paillier [32] crypto system is used as our AHE scheme, and finally ElGamal [45] is used as the MHE scheme.

4.7 Evaluation

In this section, we evaluate STYX using standard benchmarks and use cases. Our goal is to understand the overhead and thus feasibility of the system, and to estimate the efficacy of the heuristics presented in Section 4.5.

Table 4.3.: Description of continuous queries used for smart meter analytics

#	Query	Output
Q1	Number of readings	Total number of readings for the given time window
Q2	Consumption	Sum of total resource consumption for the given time window
Q3	Peak consumption	Sorted list of the aggregate consumption per 10 seconds in the given window
Q4	Top consumers	List of the distinct consumers, sorted by their total (monthly) consumption
Q5	Consumption series	Time-series of aggregate consumption per 10 seconds in the given window
Q6	Billing	Monthly bill for each consumer based on the time of usage

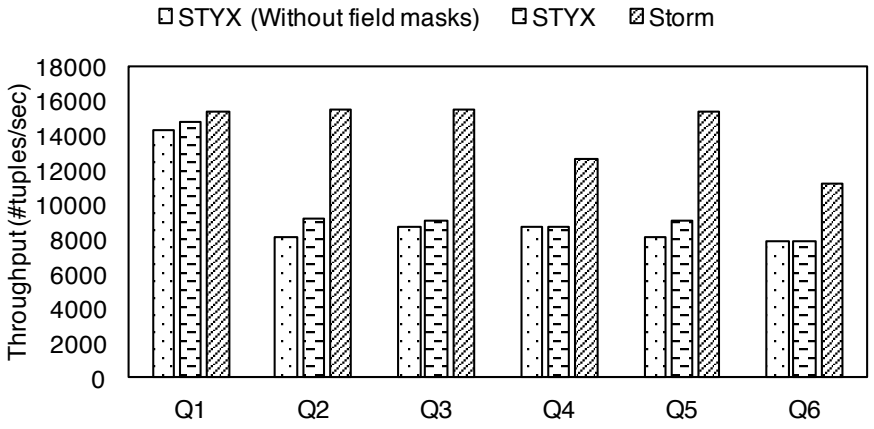
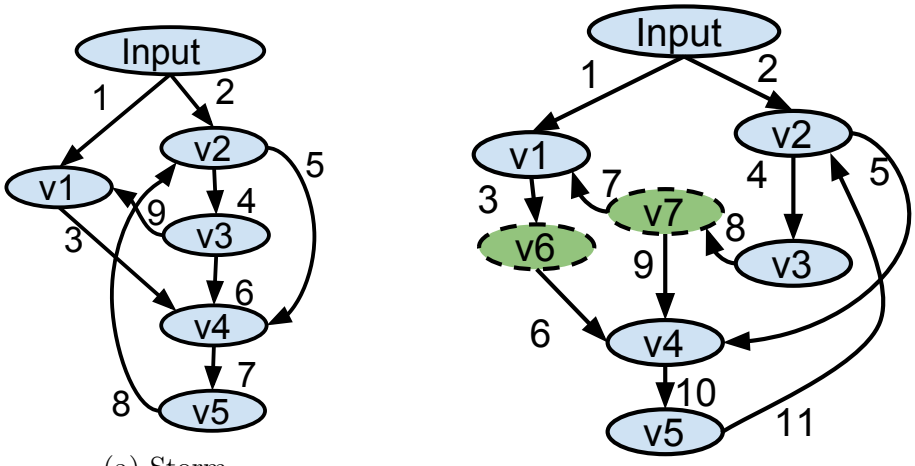


Fig. 4.5.: Smart meter query throughput

4.7.1 Smart Meter Analytics

In this evaluation we study the *throughput* of STYX by running a set of analytical queries to analyze electricity usage of homes. We use the Smart * dataset [60] available at [61] as our input. This dataset represents electrical meter readings collected over a 24-hour period at the rate of 1 reading per minute from 443 unique homes, totaling 637526 records. Each reading is a tuple of three fields: timestamp, meter id and meter reading. We define throughput as the number of tuples processed by the application graph in unit time. The runtime was configured to not drop tuples using a fixed sized

queue. When a queue becomes full, the source vertexes will stop emitting tuples. When slots in the queue free up, the vertex starts emitting tuples again. To measure throughput, we adapted the queries used in IoTBench [62] for streaming systems. Details of queries are given in Table 4.3. We used a time window of 60 seconds. To run continuous queries for at least 600 seconds, we repeated the same dataset but adjusted the timestamps. We ran these queries on 4 *large* nodes on Amazon EC2. For STYX, one of the 4 nodes was specified as a trusted node. The bandwidth of the trusted node was throttled to 8 Mbit/s to simulate a wide area network link. Results of our evaluation are presented in Figure 4.5. Q1 simply counts the number of readings and performs at 96% throughput of plain text stream. Queries Q2 to Q6, all perform Paillier addition and execute between 59% and 70% of plaintext throughput. The results also show the effect of field masking. For Q1, Q2, Q3 and Q5, we are able to mask one field, resulting in an average of 7% increase in throughput. Since queries Q4 and Q6 use all the fields in the stream, no fields could be masked.



(a) Storm

(b) STYX

Fig. 4.7.: LRB graph

4.7.2 Linear Road Benchmark

We use the Linear Road Benchmark (LRB) [63] that models variable toll calculation for a city or county and is quite widespread in the evaluation of stream processing systems. LRB simulates vehicles traveling through an expressway with vehicles generating position reports at fixed time intervals. Position reports contain information like expressway identifier, direction of travel, lane of travel, mile marker, offset within the mile, etc. These position reports are processed by a toll levying agency (e.g., city, county) to dynamically: (i) calculate the amount of toll to be levied on the vehicle, and (ii) identify accident locations in order to alert vehicles upstream of the accident etc. LRB also specifies latency invariants like time within which a toll must be calculated and the time within which an accident has to be identified. The upper limit within which the system needs to report tolls and accidents is 5 seconds. The benchmark rates the system by the highest number of expressways *(L)* the system can support while maintaining these invariants. Figure 4.6a shows the Storm topology which implements the standard linear road and Figure 4.6b shows the transformed STYX topology. Note that Figure 4.6b contains two new vertexes $v6, v7$ which are re-encryption vertexes that are executed within the trusted tier. We ran the experiment for three hours, and the rate at which position reports are emitted for one single expressway is shown in Figure 4.8. Observe that the rate of input steadily increases up to 1811 tuples per second.

LRB baseline and hypothesis validation. We first ran a baseline deployment of LRB by assigning each vertex a single task. This allows us to observe each individual vertex to see how they consume resources and verify the hypothesis made in Section 4.5.1 that *bottlenecks change when running on encrypted data streams*. We plot utilization (cf. Section 4.5) against time for the duration of a LRB run (10784 seconds) on Storm with plain text data (see Figure 4.9) and on STYX with encrypted data (Figure 4.10). We can observe that in Figure 4.9 vertexes $v4$ and $v2$ have the highest utilization values until around the 8000s mark, and after that vertex $v1$ be-

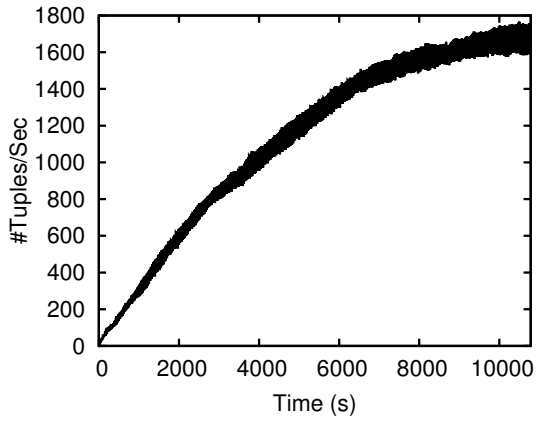


Fig. 4.8.: LRB data profile

comes the node with the highest load. This increase is because the number of tuples that requires a toll notification increases substantially after 8000s. In the transformed STYX graph running on encrypted streams, $v5$ and $v1$ come under high load until 8000s, and after that $v1$ becomes the primary bottleneck. This validates our hypothesis that primary bottlenecks differ between graphs running on plain text vs encrypted streams.

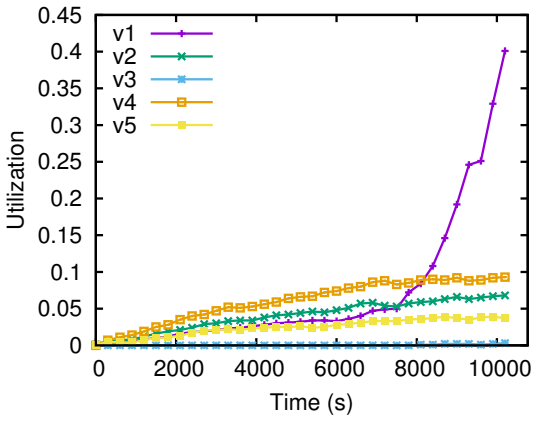


Fig. 4.9.: Storm LRB baseline

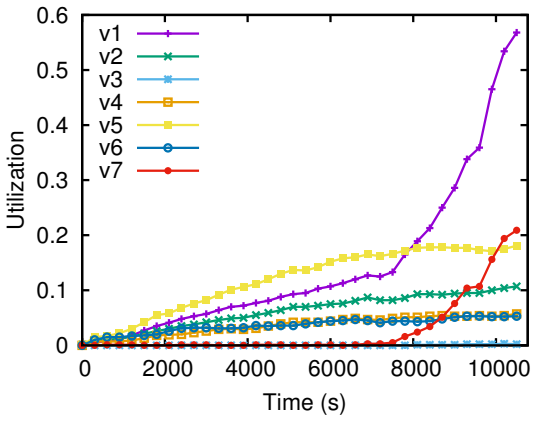


Fig. 4.10.: STYX LRB baseline

Performance of STYX deployment profile. Now we benchmark both the Storm topology graph and the transformed STYX graph using LRB. For this, we deployed

both graphs on 15 large nodes in Amazon EC2 in the best possible configuration so that the maximum number of highways supported can be identified. Table 4.4 show the results. For plain text streams Storm supports 20 expressways, while using STYX with encrypted streams we can support 15 expressways. We also plot response times for all notification triggering tuples – times taken for notifications to be issued from the time respective tuples enter the system. The response times for STYX are shown in Figure 4.12 and the response times for Storm are shown in Figure 4.11. Response times for STYX peak faster than Storm, but for 15 expressways STYX is able to maintain the response time below the threshold allowed by the benchmark.

Table 4.4.: LRB comparison

System	L	Time (ms)[1]	Profile[2]
Storm	20	2694.44	5, 4, 1, 3, 2
STYX	15	2672.97	5, 2, 1, 2, 3, 1, 1

[1] Average response time.
[2] Deployment profile for vertexes in order $v1$, ..., $v5$ for Storm and $v1$, ..., $v7$ for STYX.

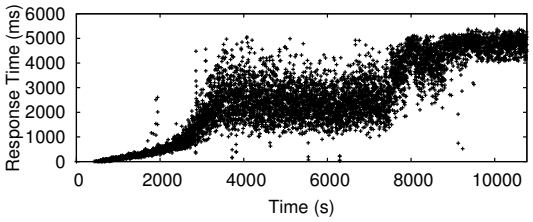


Fig. 4.11.: Response time for LRB on Storm

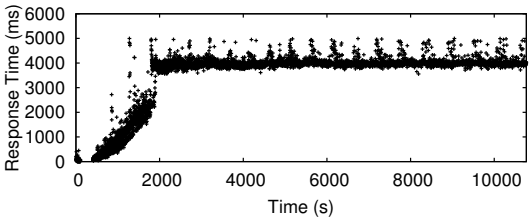


Fig. 4.12.: Response time for LRB on STYX

Effectiveness of analytical model. The effectiveness of the model can be evaluated by looking at how well the model converts the deployment profile for the plain text streams to the deployment profile for the encrypted streams in STYX. Referring back to Figures 4.9 and 4.10, these graphs can also be used as a baseline to understand our model from Section 4.5.1. Vertexes with higher utilization value should get more

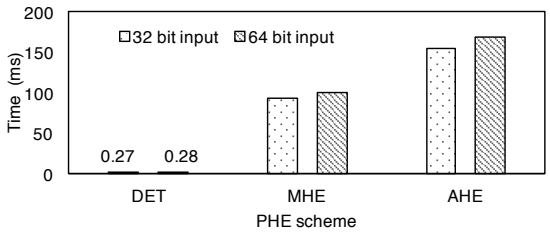


Fig. 4.13.: Encryption overhead for an IoT device

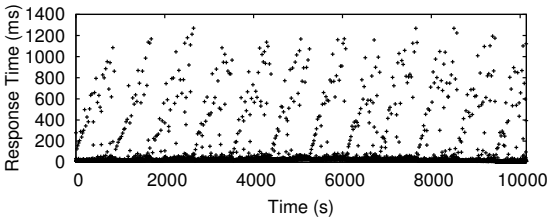


Fig. 4.14.: Response time of top-10 taxi route query with key change at the start of every month

instances to execute them. Table 4.5 shows the response time of STYX deployment profile of the Storm graph and corresponding STYX graph. As can be seen, the deployment profile generated by STYX results in the lowest response time. This profile is also in accordance with Figure 4.10 which shows vertexes $v1$ and $v4$ should get the highest numbers of instances.

Table 4.5.: LRB deployment profile response time

STYX deployment profile	Response time (ms)
5, 2, 1, 2, 3, 1, 1[1]	2672.97
4, 2, 1, 4, 2, 1, 1	2714.30
5, 4, 1, 3, 2, 1, 1	2781.40

[1] Deployment profile generated by STYX.

4.7.3 Encryption Overhead of IoT Device

We chose Raspberry Pi [64] to evaluate the overhead of encryption on IoT devices. Raspberry Pi is a widely popular IoT device and has the capabilities similar to devices found in highway cameras, vehicle tracking, and telemetry devices. We implemented AHE, MHE, and DET for Raspberry Pi version B. We measured the latency to encrypt 16 bytes, a typical message size. Our results are shown in Figure 4.13. The encryption latency is acceptable, as it is less than 150ms even for complex schemes like AHE and MHE.

4.7.4 Application Case Studies

New York taxi statistics. This application finds the top 10 most frequent routes during the last 30 minutes of taxi servicing. A route is represented by a starting grid cell and an ending grid cell. The data for this application is based on a data set released under FOIL (Freedom of Information Law) and publicly available [65]. The input data contains the locations (latitude and longitude) and times of passenger pick ups, MD5 digest of the medallion of the taxi that picked up the passenger, trip times and drop off locations (latitude and longitude). We use a simplified version of this data which contains passenger pick up time, drop off time, and route id. The dataset contains records that span over a one year time frame. Whenever the top 10 change, the output is appended. In order to evaluate the effect of key changes on response time, we simulate a key change at the beginning of each month. This means all data that are emitted with a time stamp within the first 30 minutes of every month will be encrypted under the old and new key. Response time is defined as the time between an input tuple that triggers a change in top 10 enters the system, and when the top 10 corresponding to that tuple is output. We deployed this application on 10 large nodes on Amazon EC2. Table 4.6 summarizes the results of these runs. We can see that STYX completes processing the data with only an additional 25% time compared to the Storm running on plain text stream. Furthermore, the increase in completion time or response time caused by effecting a key change every month is minimal (about 1%). Figure 4.14 shows the response time for the full run with key changes every month. In this plot, we can see intermittent spikes (total of 12) in response time for some tuples around the time a key change is in progress, but the majority of tuples (90th percentile within 31ms and 99th percentile within 818ms) respond with the same response time as when no change was in effect.

Heartbeat analysis. Finally, we study how STYX can be used for a realistic online application like a heart beat monitor. The end user application runs on specialized hardware (the monitoring device) that is capable of fast encryption and decryption.

Table 4.6.: Top-10 taxi routes

System	Completion time (s)	Average response time (ms)
Storm	8106	36.05
STYX[1]	10039	45.10
STYX[2]	10140	46.61

[1] Entire stream emitted under same key.
[2] Stream emitted with a new key every month.

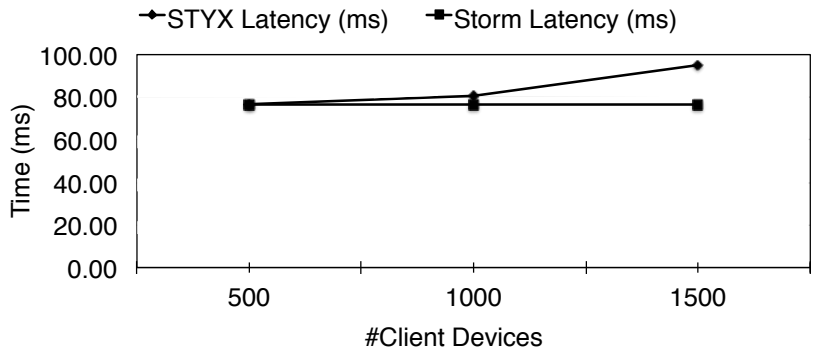


Fig. 4.15.: Heartbeat analysis response time

The monitoring device counts the number of heartbeats per minute, encrypts this value and sends it to the cloud for processing and storing. The graph running in the cloud keeps track of daily, weekly, monthly and yearly statistics. The end user may request to see these statistics on their device, in which case the data is retrieved from the topology and shown to the end user. The statistics are maintained by two vertexes, a "per user" vertex ($v1$) and an "all users" vertex ($v2$). User statistics are distributed across the multiple instances of $v1$. $v1$ also emits a summary of its per user statistics every minute which is grouped by week, month, or year by $v2$ to find the average value across all users. The client's device emits a message every time the client requests to see a specific data point. For this application, the most critical metric is the response time, that is the time a user has to wait after requesting to see a metric until the metric is actually displayed. The results of this evaluation are presented in Figure 4.15. As can be seen, the response times for STYX are

very close to the plain text version for up to 1000 client devices after which STYX's response time degrades compared to the plain text stream running on Storm. This demonstrates that for applications where the input rate is not very high, we are able to get response times similar to plain text streams.

5. RELATED WORK

This chapter presents work related to this thesis. The first section present works related to integrity and availability and the second section present works related to confidentiality.

5.1 Integrity and Availability

Systems providing integrity. File systems like SUNDR [66], Sirius [67], CloudProof [68], and Caelus [69] ensure integrity of data stored in them, but do not verify correctness of computation results. This means a byzantine computation can produce incorrect results that may be saved in these file systems. An alternate approach to integrity is enforcing data flow or control flow integrity in programs. Work done by Castro et al. [70] computes the data-flow graph of a program using static analysis. The program is then instrumented to ensure that the runtime flow matches the statically computed data-flow. Systems like SIF [71], [72], Urflow [73] and Resin [74] use this approach. Control flow integrity protection mechanisms like [75] follow the same approach, but for control flow. However, as pointed out by [76], attacks can succeed without changing the control-flow of the target programs. Further, both approaches do not protect against byzantine faults in compute nodes. Verna [77] is a web application platform that provides end-to-end integrity guarantees against attackers with full access to the web and database servers. Verena uses authenticated data structures [78–80] to verify the integrity of a web page by verifying the results of queries on data stored at the server.

Systems providing BFT. Works like PBFT [81, 82], Q/U protocol [83] and HQ Replication [84] show how to make BFT replication practical in general. Libraries like UpRight [85], BFT-SMaRt [26] and EBAWA [86] make it practical for anyone to

efficiently and quickly implement some of these systems. Recent work like Zyzzyva [87] (based on Fast Paxos [88,89]) further improve the performance and efficiency of some of these solutions. All these solutions focus on replicating monolithic servers and do not provide parametrizable tradeoffs between overhead and fault tolerance. Yin et al. [19] separate request ordering from request execution in BFT server replication; we separate our architecture based on differences in the guarantees offered by nodes and not for consistency (no mutable shared state).

BFT in cloud. With respect to cloud-based computations, Byzantine Fault Tolerant Mapreduce [90] explores executing Byzantine fault tolerant MapReduce jobs in the cloud and tries to reduce overhead by only starting $f + 1$ replicas of map and reduce tasks. Byzantine fault tolerance is achieved by restarting map and reduce tasks if $f + 1$ replicas do not agree on the output. This reduces the overhead when failures are not frequent but does not reduce the number of synchronization points required during job execution. BFTCloud [91] tries to secure generic computations run on voluntary resource clouds, but does not look at data-flow job specific optimizations. BFTHadoop [90] applies BFT replication to all sub-components involved in a Hadoop MapReduce job. Most of these solutions explore adapting BFT solutions to the cloud, but ClusterBFT differs from them in the following ways. None of these solutions try to reduce the number of verification instances required, or to actively identify faulty nodes by overlapping job clusters. ClusterBFT also provides parametrizable tradeoffs between overhead and performance.

Verifiability. Works like Pepper [92] and Ginger [93] show that output verifiability [94,95] is becoming more practical. These systems allows the computation initiator to encode the computation in such a way that it is possible to verify the result using the computation output and key. Pinocchio [96] further reduces the overhead involved and allows public verifiability. Even with these considerable improvements, these systems incur an overhead that is linearly proportional to the complexity of the computation. These systems are also limited with respected to computations involv-

ing dynamic looping constructs; requiring the programmer to inform the compiler how far the loop should be unrolled.

5.2 Confidentiality

Approaches to protect confidentiality of data vary based on system and threat models. Differential privacy [97] aims to improve the accuracy of statistical queries, without revealing information about individual records. The server which performs the query in this case is trusted with access to plain text data. This is in contrast to SPR that assumes data and computation reside in an untrusted server. sTile [98] keeps data confidential by distributing computations onto large numbers of nodes in the cloud, requiring the adversary to control more than half of the nodes in order to reconstruct input data. SPORC [99] and Depot [100] are systems that run mostly on the clients, without having to trust a server. Airavat [101] combines mandatory access control and differential privacy to enable MapReduce computations over sensitive data. Both sTile and Airavat require the cloud provider and cloud infrastructure to be trustworthy. Excalibur [102] utilizes trusted platform modules to ensure cloud administrators cannot modify or examine data. Compared to SPR, Excalibur adds extra cost because the trusted platform modules need to be installed and maintained. However, SPR does not guarantee integrity. Several systems [103, 104] focus on transparently encrypting and decrypting data transmitted to a server. Like mentioned above for file systems, these systems also cannot compute on encrypted data.

Program analysis. MrCrypt [35] consists in a program analysis for MapReduce jobs that tracks operations and their requirements in terms of PHE. When sequences of operations are applied to a same field, the analysis defaults to FHE, noting that the system does not currently execute such jobs at all due to lack of available FHE cryptosystems. Thus several PigMix benchmarks are incompletely covered for evaluation. Work has also been done on secure programming languages for distributed systems. These include languages like DSL [105] and SMCL [106], which include type

systems that ensure that programs run securely. However, these languages focus on other issues such as interaction between different users, and (thus) do not support automatic parallelization via MapReduce or a similar framework.

Systems that process encrypted data. CryptDB [34] is a seminal system leveraging PHE for data management. As mentioned earlier CryptDB is a database system that stores encrypted data and executes SQL queries directly on encrypted data. CryptDB uses a trusted proxy that analyzes queries and decrypts columns to an encryption level suitable for query execution. It does not consider the more expressive kind of data flow programs as in Pig Latin, or MapReduce-style parallelization of queries. Monomi [56] improves performance of encrypted queries similar to those used in CryptDB and introduces a designer to automatically choose an efficient design suitable for each workload. Seabed [107] introduces a new symmetric key cryptosystem that is partially homomorphic with respect to addition to perform data analysis. Bost et al. [108] introduce algorithms for performing machine learning classification on encrypted data. Both these are orthogonal to our work and these new cryptosystems can be easily integrated with SPR to reduce execution overheads. Mylar [109] is a platform for building web applications, which protects data confidentiality by ensuring that the web server only sees encrypted data. Data is decrypted only in the client's browser. This protects web applications from attackers with full access to servers. Mylar is more suited for stateless web applications and not suited to perform typical data analysis queries.

Stream processing systems. To the best of our knowledge, STYX is the first system to address the issue of data confidentiality in the context of continuous query execution by successfully putting PHE to work. A large body of work on stream and data flow processing has addressed many challenges related to stream processing such as fault tolerance, continuous query processing and analytics (e.g., Aurora project [110–112], and Telegraph project [113, 114]). With the advent of cloud computing, the need for highly scalable stream processing systems has resulted in the next generation of stream processing systems such as Storm [59], Heron [115], Spark

streaming [116], and Samza [117]. STYXs design is based on the stream processing design of Storm; our concepts can also be implemented on top of other stream processing systems. Systems line CryptDB and Monomi discussed above follow a centralized database design and do not consider streaming workloads as supported by STYX. As such it is unclear whether CryptDB or Monomi can effectively process stream data workloads.

Trusted hardware. An alternate approach to ensure confidentiality is to use specially designed hardware [118–122]. Hardware security modules (HSMs) [123], for example can be used to store secret keys in the cloud. AWS CloudHSM [124] offer APIs for securely storing and manipulating secret keys. This approach can protect secret keys, but user data still needs to be in plain text for computations to happen. Further, HSMs and related services are expensive. Another specialized hardware that is becoming popular now is Intel SGX [125]. SGX offers hardware encrypted and integrity protected physical memory, which allows data and code to reside in the untrusted cloud. SGX introduces a set of new instructions that allows a process to create a secure region within the main memory known as an enclave. Haven [126] protects unmodified legacy applications by running applications within a small library OS that executes inside an SGX enclave. CipherBase [127] provides an FPGA-based implementation of a trusted hardware that can be used to run a commercial SQL database system without sacrificing data confidentiality. TrustedDB [128] preserves confidentiality by using a server-hosted, tamper-proof trusted hardware in critical query processing stages. These approaches require the end user to trust another party (Intel for SGX). Trusting another party may not always be practical because of government regulations or policies. Relying only on trusted hardware also means that we are unable to utilize the compute capacity that is currently available in the form of commodity hardware. These approaches can also be used complimentary to our approach by allowing secure computations or sub-computations to be performed on a trusted hardware in an untrusted cloud.

6. CONCLUSION

In this chapter we summarize our work and describe future directions.

6.1 Summary

We presented the design and evaluation of three systems; ClusterBFT, a system for assured data processing and analysis, SPR, a system able to perform big data analysis jobs on encrypted data and STYX, a distributed system for evaluating continuous queries over encrypted data streams. ClusterBFT achieves its objectives with practical overheads by using variable-degree clustering, approximated output comparison and fault isolation.

SPR leverages data flow analysis and program transformation to perform big data analysis jobs on encrypted data, which allows us to leverage untrusted cloud resources while at the same time preserving confidentiality. Our program transformations translate Pig Latin scripts into semantically equivalent scripts that operate entirely on encrypted data. By automatically transforming Pig Latin scripts, developer efforts for achieving security are minimized. We evaluated our approach through standard benchmarks and applications and demonstrated its applicability and performance. Our evaluations show that our approach scales well and can handle big volumes of encrypted data whilst retaining good performance. Furthermore, we gauge the savings in terms of developer effort of our automatic approach by comparing scripts that operate on unencrypted data with the transformed Pig Latin script which our system generates.

Finally, our system to execute continuous queries, STYX also leverages untrusted public cloud resources as its compute platform without sacrificing confidentiality. STYX exposes an API which allows programmers to develop secure applications with

little or no knowledge of the underlying crypto systems and STYX heuristics ensure a deployment that optimizes cloud usage. Our evaluations show that we can meet the latency requirements even when the volume of encrypted traffic is high.

6.2 Future Directions

Improving cryptosystems. The latency and computation overhead of performing evaluations on encrypted operands can be reduced by improving the underlying partially homomorphic cryptosystems. Seabed [107] reduces the overhead of performing additions on encrypted operands by introducing a new symmetric key cryptosystem. Similarly, symmetric key cryptosystems, partially homomorphic with respect to multiplication can reduce overhead of multiplication operations. Yet another aspect of cryptosystems that can be improved is the size of encrypted operands. If we can design cryptosystems with the size of encrypted operands comparable to the size of plain text, cost of data transmission and computation can be brought down substantially.

Big data machine learning. Machine learning has become synonymous with many common tasks required as part of our daily lives. Medical or genomics predictions, spam detection, face recognition, finance, etc., all rely on machine learning. Many of these applications handle sensitive data as well. Performing these machine learning algorithms over encrypted data is one way to achieve the confidentiality requirements of such sensitive data. Bost et al., [108] introduce few classification techniques that can be run over encrypted data, but does not address challenges of scale caused by big data. Improvements in this direction will allow faster adoption and improve applicability of these techniques.

Program analysis. We can also investigate a number of optimizations and further heuristics for selecting from different possible execution paths in the transformed Pig Latin script. In particular, paths which perform re-encryption of data at clients (in contrast to client-side termination or costly re-encryption in the cloud as with FHE) and minimizing these re-encryptions themselves. To this end we can employ

sampling to determine amounts of data at different points in analysis jobs in order to better select between different options.

REFERENCES

REFERENCES

[1] "Department of Defense Information Enterprise Strategic Plan 2011-2012," http://dodcio.defense.gov/docs/DodIESP-r16.pdf.

[2] D. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–242, 1976.

[3] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, 1982.

[4] J. J. Stephen and P. Eugster, "Assured Cloud-Based Data Analysis with ClusterBFT," in *International Conference on Middleware (MIDDLEWARE)*, 2013, pp. 82–102.

[5] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine Fault Detectors for Solving Consensus," *The Computer Journal*, vol. 46, no. 1, pp. 16–35, 2003.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[8] "Apache Pig," http://pig.apache.org.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in *International Conference on the Management of Data (SIGMOD)*, 2008, pp. 1099–1110.

[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-clouds," in *European Conference on Computer Systems (EuroSys)*, 2011, pp. 31–46.

[11] P. Verissimo, A. Bessani, and M. Pasin, "The TClouds Architecture: Open and Resilient Cloud-of-clouds Computing," in *International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1 –6.

[12] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 1–14.

[13] "Spark," http://spark.apache.org/streaming/.

[14] MRC, "DARPA-BAA-11-55: I2O Mission-oriented Resilient Clouds (MRC)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-55/listing.html.

[15] A. Newell, D. Obenshain, T. Tantillo, C. Nita-Rotaru, and Y. Amir, "Increasing Network Resiliency by Optimally Assigning Diverse Variants to Routing Nodes," in *International Conference on Dependable Systems and Networks (DSN)*, 2013.

[16] S. Pleisch, A. Kupsys, and A. Schiper, "Preventing Orphan Requests in the Context of Replicated Invocation," in *Symposium on Reliable Distributed Systems (SRDS)*, 2003, p. 119.

[17] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing Research Agenda," *SIGACT News*, vol. 40, no. 2, pp. 68–80, Jun. 2009.

[18] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *Symposium on Operating System Design and Implementation (OSDI)*, 2006, pp. 335–350.

[19] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *Special Interest Group on Operating Systems Review (SIGOPS)*, vol. 37, no. 5, pp. 253–267, 2003.

[20] Hadoop, "Hadoop," http://hadoop.apache.org/.

[21] C. Olston and B. Reed, "Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows," in *International Conference on the Management of Data (SIGMOD)*, 2011, pp. 1221–1224.

[22] "A programmable cloud-computing research testbed," http://www.vicci.org.

[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *International Conference on World Wide Web (WWW)*, 2010, pp. 591–600.

[24] "Pig Lab," https://github.com/michiard/CLOUDS-LAB/.

[25] "Airline Data," http://stat-computing.org/dataexpo/2009/the-data.html.

[26] "High-performance Byzantine Fault-Tolerant State Machine Replication," https://code.google.com/p/bft-smart/.

[27] NCDC, "weatherdata snapshot," http://aws.amazon.com/datasets/2759.

[28] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster, "Practical Confidentiality Preserving Big Data Analysis," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[29] J. J. Stephen, S. Savvides, R. Seidel, and P. T. Eugster, "Program analysis for secure big data processing," in *International Conference on Automated Software Engineering (ASE)*, 2014, pp. 277–288.

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *European Conference on Computer Systems (EuroSys)*, 2007, pp. 59–72.

[31] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Annual International Cryptology Conference (CRYPTO)*, vol. 1, Aug. 2013, pp. 75–92.

[32] P. Paillier, "Public-key Cryptosystems Based on Composite Degree Residuosity Classes," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 1999, pp. 223–238.

[33] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, Efficient Data-parallel Pipelines," in *Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 363–375.

[34] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Symposium on Operating System Principles (SOSP)*, 2011, pp. 85–100.

[35] S. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static Analysis for Secure Cloud Computations," in *Conference on Object-Oriented Programming Systems Language and Applications (OOPSLA)*, 2013, pp. 271–286.

[36] "Apache PigMix benchmark," https://cwiki.apache.org/confluence/display/PIG/PigMix.

[37] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[38] I. DeveloperWorks, "Process your Data with Apache Pig," 2012, http://www.ibm.com/developerworks/library/l-apachepigdataquery/.

[39] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-Preserving Symmetric Encryption," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2009, pp. 224–241.

[40] "Zero MQ," http://zeromq.org.

[41] "The GNU Multiple Precision Arithmetic Library," https://gmplib.org/.

[42] B. Schneier, "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)," in *Fast Software Encryption*, 1994, pp. 191–204.

[43] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – the Advanced Encryption Standard*, 2002.

[44] S. Halevi and P. Rogaway, "A Tweakable Enciphering Mode," in *Annual International Cryptology Conference (CRYPTO)*, 2003, pp. 482–499.

[45] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[46] "Amazon EC2," http://amazon.com/ec2.

[47] "HElib," https://github.com/shaih/HElib.

[48] "Wikipedia Database Download," http://en.wikipedia.org/wiki/Wikipedia:Database_download.

[49] M. Schwarzkopf, D. Murray, and S. Hand, "The Seven Deadly Sins of Cloud Computing Research," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[50] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster, "STYX: Stream Processing with Trustworthy Cloud-based Execution," in *Symposium on Cloud Computing (SoCC)*, 2016.

[51] "SmartThings," http://www.smartthings.com/.

[52] "Nest," https://nest.com/.

[53] J. Groopman and S. Etlinger, "Consumer Perceptions of Privacy in The Internet of Things," Tech. Rep., 2015.

[54] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data Management for Connected Homes," in *Symposium on Networked System Design and Implemtation (NSDI)*, 2014, pp. 1–14.

[55] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic Evaluation of the AES Circuit," in *Annual International Cryptology Conference (CRYPTO)*, 2012, pp. 850–867.

[56] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing Analytical Queries over Encrypted Data," *Proceedings of VLDB Endowment*, vol. 6, no. 5, pp. 289–300, 2013.

[57] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, and W. Hu, "Talos: Encrypted Query Processing for the Internet of Things," in *Conference on Embedded Networked Sensor Systems (SenSys)*, 2015, pp. 197–210.

[58] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-Preserving Symmetric Encryption," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2009, pp. 224–241.

[59] "Apache Storm," https://storm.apache.org/.

[60] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht, "Smart*: An Open Dataset and Tools for Enabling Research in Sustainable Homes," *Workshop on Data Mining Applications in Sustainability, (SustKDD)*, 2012.

[61] "Smart* Data Set for Sustainability," http://traces.cs.umass.edu/index.php/Smart/Smart.

[62] M. F. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver, "IoTAbench: an Internet of Things Analytics Benchmark," in *International Conference on Performance Engineering (ICPE)*, 2015, pp. 133–144.

[63] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear Road: A Stream Data Management Benchmark," in *International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 480–491.

[64] "Raspberry Pi," 2012, https://www.raspberrypi.org.

[65] "NYCs Taxi Trip Data," http://chriswhong.com/open-data/foil_nyc_taxi/.

[66] J. Li, M. N. Krohn, D. Mazières, and D. E. Shasha, "Secure Untrusted Data Repository (SUNDR)," in *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[67] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing Remote Untrusted Storage," in *Network and Distributed System Security Symposium (NDSS)*, 2003.

[68] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof," in *USENIX Annual Technical Conference (ATC)*, 2011.

[69] B. H. Kim and D. Lie, "Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices," in *IEEE Symposium on Security and Privacy (SP)*, 2015.

[70] M. Castro, M. Costa, and T. L. Harris, "Securing Software by Enforcing Data-flow Integrity," in *Symposium on Operating System Design and Implementation (OSDI)*, 2006, pp. 147–160.

[71] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing Confidentiality and Integrity in Web Applications," in *USENIX Security Symposium*, 2007.

[72] W. K. Robertson and G. Vigna, "Static Enforcement of Web Application Integrity Through Strong Typing," in *USENIX Security Symposium*, 2009, pp. 283–298.

[73] A. Chlipala, "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications," in *Symposium on Operating System Design and Implementation (OSDI)*, 2010, pp. 105–118.

[74] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving Application Security with Data Flow Assertions," in *Symposium on Operating System Principles (SOSP)*, 2009, pp. 291–304.

[75] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[76] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *USENIX Security Symposium*, 2005.

[77] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, "Verena: End-to-End Integrity Protection for Web Applications," in *IEEE Symposium on Security and Privacy (SP)*, 2016.

[78] S. A. Crosby and D. S. Wallach, "Authenticated Dictionaries: Real-World Costs and Trade-Offs," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 2, p. 17, 2011.

[79] J. L. Muñoz, J. Forné, O. Esparza, and M. Soriano, "Implementation of an Efficient Authenticated Dictionary for Certificate Revocation," in *IEEE Symposium on Computers and Communications (ISCC)*, 2003, pp. 238–243.

[80] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic Authenticated Index Structures for Outsourced Databases," in *International Conference on the Management of Data (SIGMOD)*, 2006, pp. 121–132.

[81] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Symposium on Operating System Design and Implementation (OSDI)*, 1999.

[82] ——, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[83] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine Fault-tolerant Services," in *Symposium on Operating System Principles (SOSP)*, 2005, pp. 59–74.

[84] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance," in *Symposium on Operating System Design and Implementation (OSDI)*, 2006, pp. 177–190.

[85] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright Cluster Services," in *Symposium on Operating System Principles (SOSP)*, 2009, pp. 277–290.

[86] G. Santos Veronese, M. Correia, A. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks," in *International Symposium on High-Assurance Systems Engineering (HASE)*, 2010, pp. 10 –19.

[87] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," in *Symposium on Operating System Principles (SOSP)*, 2007, pp. 45–58.

[88] L. Lamport, "Future Directions in Distributed Computing," 2003, ch. Lower Bounds for Asynchronous Consensus, pp. 22–23.

[89] P. Dutta, R. Guerraoui, and M. Vukoli, "Best-case Complexity of Asynchronous Byzantine Consensus," Tech. Rep., 2005.

[90] P. Costa, M. Pasin, A. Bessani, and M. Correia, "Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011, pp. 32 –39.

[91] Y. Zhang, Z. Zheng, and M. R. Lyu, "BFTCloud: A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing," *International Conference on Cloud Computing*, vol. 0, pp. 444–451, 2011.

[92] A. J. B. S. Setty, R. McPherson and M. Walsh, "Making Argument Systems for Outsourced Computation Practical (sometimes)," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[93] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking Proof-based Verified Computation a few Steps Closer to Practicality," in *Conference on Security Symposium (Security)*, 2012, pp. 12–12.

[94] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge," in *Annual International Cryptology Conference (CRYPTO)*, 2013, pp. 90–108.

[95] M. Walfish and A. J. Blumberg, "Verifying Computations without Reexecuting Them: from Theoretical Possibility to Near-Practicality," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 20, p. 165, 2013.

[96] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly Practical Verifiable Computation," *IACR Cryptology ePrint Archive*, vol. 2013, p. 279, 2013.

[97] C. Dwork and K. Nissim, "Privacy-Preserving Datamining on Vertically Partitioned Databases," in *Annual International Cryptology Conference (CRYPTO)*, 2004, pp. 528–544.

[98] Y. Brun and N. Medvidovic, "Keeping Data Private while Computing in the Cloud," in *International Conference on Cloud Computing (CLOUD)*, 2012, pp. 285–294.

[99] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group Collaboration using Untrusted Cloud Resources," in *Symposium on Operating System Design and Implementation (OSDI)*, 2010, pp. 337–350.

[100] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage with Minimal Trust," in *Symposium on Operating System Design and Implementation (OSDI)*, 2010, pp. 307–322.

[101] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for MapReduce," in *Symposium on Networked System Design and Implemtation (NSDI)*, 2010.

[102] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services," in *Conference on Security Symposium (SECURITY)*, 2012, pp. 10–10.

[103] F. Beato, M. Kohlweiss, and K. Wouters, "Scramble! Your Social Network Data," in *Privacy Enhancing Technologies Symposium (PETS)*, 2011, pp. 211–225.

[104] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao, "Silverline: Toward Data Confidentiality in Storage-Intensive Cloud Applications," in *Symposium on Cloud Computing (SoCC)*, 2011, p. 10.

[105] J. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "Information-flow Control for Programming on Encrypted Data," in *Computer Security Foundations Symposium (CSF)*, June 2012, pp. 45–60.

[106] J. D. Nielsen and M. I. Schwartzbach, "A Domain-specific Programming Language for Secure Multiparty Computation," in *Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2007, pp. 21–30.

[107] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big Data Analytics over Encrypted Datasets with Seabed," in *Symposium on Operating System Design and Implementation (OSDI)*, 2016.

[108] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine Learning Classification over Encrypted Data," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[109] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan, "Building Web Applications on Top of Encrypted Data Using Mylar," in *Symposium on Networked System Design and Implemtation (NSDI)*, 2014.

[110] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *Biennial Conference on Innovative DataSystems Research (CIDR)*, vol. 3, 2003, pp. 257–268.

[111] J. H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-availability Algorithms for Distributed Stream Processing," in *International Conference on Data Engineering (ICDE)*, 2005, pp. 779–790.

[112] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis Distributed Stream Processing System," in *International Conference on the Management of Data (SIGMOD)*, 2008, pp. 13–24.

[113] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *Biennial Conference on Innovative DataSystems Research (CIDR)*, vol. 20, no. March, 2003, p. 668.

[114] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly Available, Fault-tolerant, Parallel Dataflows," in *International Conference on the Management of Data (SIGMOD)*, 2004, pp. 827–838.

[115] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *International Conference on the Management of Data (SIGMOD)*, 2015, pp. 239–250.

[116] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012, pp. 10–10.

[117] "Samza," http://samza.apache.org/.

[118] D. Champagne and R. B. Lee, "Scalable Architectural Support for Trusted Software," in *International Conference on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[119] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME: A Hardware-Software Approach to Full System Security," in *International Conference on Supercomputing (ICS)*, 2011, pp. 108–119.

[120] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. S. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 2–13.

[121] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000, pp. 168–177.

[122] E. Owusu, J. Guajardo, J. M. McCune, J. Newsome, A. Perrig, and A. Vasudevan, "OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms," in *ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 13–24.

[123] S. W. Smith and S. H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor," *Computer Networks*, vol. 31, no. 8, pp. 831–860, 1999.

[124] "AWS CloudHSM," https://aws.amazon.com/cloudhsm/.

[125] "Intel SGX," https://software.intel.com/en-us/isa-extensions/intel-sgx#.

[126] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Symposium on Operating System Design and Implementation (OSDI)*, 2014, pp. 267–283.

[127] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "Orthogonal Security with Cipherbase," in *Biennial Conference on Innovative DataSystems Research (CIDR)*, 2013.

[128] S. Bajaj and R. Sion, "TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality," in *International Conference on the Management of Data (SIGMOD)*, 2011, pp. 205–216.

VITA

VITA

Julian James Stephen was born in the city of Trivandrum, India. He received a BTech in computer science with distinction from Mahatma Gandhi University, Kottayam, India in 2004. Julian worked as a software engineer at Infosys from 2004 to early 2006, after which he worked for Oracle as an application engineer till 2010. He joined Purdue University to pursue graduate studies and he received an M.S. in computer science and Ph.D under that direction of Dr. Patrick Eugster in 2016. Julian is interested in building systems and developing programming abstractions that make distributed computing easier. His research areas include distributed systems, confidential data analytics and software engineering. During his Ph.D. studies, Julian also did three internships at HP Labs.